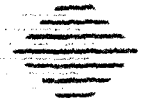


Carnegie Mellon University
Software Engineering Institute

Formal Specification and Verification of Concurrent Programs

Curriculum Module SEI-CM-27-1.0

Formal Specification and Verification of Concurrent Programs

SEI Curriculum Module SEI-CM-27-1.0

February 1993

Daniel M. Berry
Technion and Software Engineering Institute

Carnegie Mellon University
Software Engineering Institute

This work was sponsored by the U.S. Department of Defense.
Approved for public release. Distribution unlimited.

This technical report was prepared for the

SEI Joint Program Office
ESC/AVS
Hanscom AFB, MA 01731

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

Review and Approval

This report has been reviewed and is approved for publication.

FOR THE COMMANDER

Thomas R. Miller, Lt Col, USAF
SEI Joint Program Office

The Software Engineering Institute is sponsored by the U.S. Department of Defense.
This report was funded by the U.S. Department of Defense.

Formal Specification and Verification of Concurrent Programs

Acknowledgements

Thanks to Nancy Leveson, Bob Glass, and Norm Gibbs for pushing me to write this module and thanks to Mark Ardis, Allison Brunvand, Dave Bustard, Lionel Deimel, Gary Ford, Linda Pesante, and Mary Zoys for their detailed assistance in its writing. Thanks to Pamela Zave and Jeannette Wing for reading parts or all of earlier drafts of this module; their comments were helpful in getting the module to its present state.

Contents

Capsule Description	**1**
Philosophy	**1**
Objectives	**2**
Prerequisite Knowledge	**2**
Module Content	**3**
Outline	**3**
Annotated Outline	**4**
Glossary	**46**
Teaching Considerations	**49**
Suggested Schedules	**49**
Worked Examples	**49**
Exercises	**49**
Caveats	**51**
Bibliography	**53**
Tables and Figures	**81**

Formal Specification and Verification of Concurrent Programs

Capsule Description

This module introduces formal specification of concurrent software and verification of the consistency between concurrent programs and their specifications. First, what one might want to be able to prove about a concurrent program is discussed. Then, a number of formal descriptions of the concept are presented. These vary in their coverage of the phenomena, and some can be used as the bases of formal specifications of programs. Next, techniques for carrying out the proof of consistency between the specification and the program are described. Finally, it is noted that some of these techniques have automated tools such as verifiers associated with them.

Philosophy

Programming concurrent software is a complex, error-prone task. Because of the inherent nondeterminism, it is difficult for the programmer to understand the effect of his or her own program. It is even more difficult for others, such as the client and the maintainer, to understand this effect.

Formal specification of concurrent software and verification of the consistency of the software with respect to these specifications are useful if for no other reason than they force a closer examination of the software. Sometimes, the formal models exhibit aspects of the nondeterministic behavior that were not otherwise apparent. Other times, just the plain fact of redundancy—the specification and the program are two statements of the same thing, but in different languages—is what is useful.

It is clear that the cost of carrying out formal specification and consistency verification is high. It is so high that formal specification and verification will not be used in a software development unless they are perceived as reducing the total cost of the development. That is, they must be perceived as reducing the number of eventual errors, and the perceived cost of the residual errors were they not done must be higher than the cost of carrying them out.

Consistent with this utilitarian view is the observation that very often the process of writing the formal specification of a system is the same as the process of designing the system's functionality. That is, the act of writing the formal specification is simply recording the requirement decisions that have been made, and most changes taking place in a specification before the first verification attempt are made for the purpose of getting the function of the system right.

This philosophy dictates what material is included in this module. Material is included if, in the opinion of the author, it is oriented toward the practice of software development, that is, if the author believes that the material can be used to help the software engineer develop systems or applications that exhibit concurrency. Hence material describing development methods and specification and verification environments is included. Theoretical work is described to the extent that it provides the logical basis for practical work. Deep theoretical issues—such as axiomatic completeness, formal modeling of fairness, which are important in their own right—are not covered here because they do not have an impact on the applicable work.

Another issue dictating what is covered herein is the simple fact that as this module is being written, the field is expanding! Indeed, the release of this module has been delayed more than once by the discovery of recent new material. An arbitrary decision was taken to release the module now with what is already in it. Surely, the document is thick enough!

Finally, there is still other not-so-recent material that is consistent with the philosophy and is nevertheless not covered in more detail than a mere mentioning with a bibliographical citation. These citations point to approaches that are so similar to approaches covered in detail, that not enough would be gained by discussing them in detail. The choice of which approach to cover among similar approaches was arbitrary, reflecting what was known first to the author, and should not be construed as saying that the presented approach is any better than the others.

In any case, this module discusses neither VDM [Bekic74] nor Z [Hayes87] because

1. VDM is covered in another module [Pedersen89] in detail, and
2. neither is really intended for use in dealing with concurrency; they both aim at treating programs as functions, and most concurrent programs, being nonhalting systems, are just not describable as functions.

Objectives

The student who has absorbed the material of this module can be expected to

Know

- of the various tools and environments of tools for carrying out specification and consistency verification

Comprehend

- the basic terminology of concurrency
- the various properties that concurrent systems may or may not satisfy and the meanings of and differences between these properties
- the various formal models of concurrency and their relations to each other and their coverage
- the various formal specification languages
- the methods for proving programs consistent with specifications

Apply

- at least one of the specification languages to a problem of moderate size

Analyze

- specifications in any of the various specification languages
- verifications of consistency carried out in any of the proof systems

Synthesize

- a specification of a program in at least one specification language
- a verification of the consistency of a program to specifications in the above language

Evaluate

- the coverage of any of the above or new formal models, specification languages, and environments

Prerequisite Knowledge

The student should be fully familiar with the material of the curriculum module *Formal Verification of Programs* [Berztiss88] and of all of its prerequisites, especially those of programming and mathematical maturity. The student should also be familiar with the of the curriculum modules *Concepts of Concurrent Programming* [Bustard90] and *Languages and System Support for Concurrent Programming* [Feldman90] and their support materials.

It is useful, but not essential, to have some familiarity with the material in the curriculum modules *Formal Specification of Software* [Berztiss87], *Software Specification: A Framework* [Rombach87], and *Software Development using VDM* [Pedersen89].

Module Content

Outline

I. Unifying Model

II. Properties of Concurrent Programs
  1. Safety Properties
    a. Deadlock Freeness
    b. Mutual Exclusion
    c. Data Security
    d. Proper Termination
    e. Partial Correctness
  2. Liveness Properties
    a. Fairness
    b. Receiving Requested Data
    c. Sent Message Will Arrive
    d. Each Request Serviced
    e. Termination
    f. Total Correctness
  3. Others

III. Formal Models
  1. Operational — NDISM
    a. Description of NDISM by Program
    b. Formal Mathematical Descriptions of NDISMs
    c. Redundant Specification of Properties
    d. Formal Verification of Redundantly Specified Properties
    e. Graph Models of Concurrent Computation
  2. Axiomatic
  3. Temporal Logic
  4. Denotational
  5. Comparisons of the Coverage of the Approaches

IV. Actual Specifications of Software
  1. Operating System Security
  2. Database Integrity
  3. Protocols
  4. Other Problems

V. Doing the Verifications

VI. Specification Languages and Verification Environments
  1. AFFIRM
  2. FDM
  3. Gypsy
  4. HDM
  5. P-NUT
  6. SARA
  7. PAISLey
  8. STATEMATE
  9. Process Algebras
  10. ASTRAL

VII. Current Status

Annotated Outline

I. Unifying Model

For the purpose of unifying the discussion of all of the formal models of concurrency covered in this module and for providing a basis for comparing them with each other, this module uses the formal model of computation, known as the Nondeterministic Information Structure Model (NDISM) [Wegner68]. It is a state machine formalism in which the content of the state is left unspecified but can be made as simple or as complex as desired. The same pictorial version of this formal model is used in the textbook *Programming Language Structures* [Organick78] and in "A Visual Execution Model for Ada Tasking" [Dillon92].

In NDISMs, concurrency is modeled by supposing the existence of process items in the state and nondeterministically selecting one such process to execute one instruction at each state transition. In other words, concurrency is modeled by interleaving process computations at the granularity of the single instruction.

For the purpose of establishing the notation used in this module, the basic definitions given in the support materials are repeated in this section.

The formal model is that of a *nondeterministic information structure model* (NDISM) [Wegner68].

$M=(I, I_0, F)$ is an NDISM if and only if

1. I is a countable set
2. $I_0 \subseteq I$
3. $F: I \rightarrow \mathcal{P}(I)$

The elements of the set I are called *snapshots*.

A possibly infinite sequence $C=(s_0, \ldots, s_{i-1}, s_i, \ldots)$ is a computation in an NDISM $M=(I, I_0, F)$ if and only if

1. $\forall s_j$ in C, $s_j \in I$
2. $s_0 \in I_0$
3. $\forall i>0, s_i \in F(s_{i-1})$
4. C is a proper initial subsequence of no other sequence meeting conditions 1-3

Note that the i^{th} snapshot is one of the elements yielded by applying the *transformation* F to the $i-1^{th}$ snapshot. Herein lies the nondeterminism; in each step of each computation, the next step is chosen from among possibly several candidate next steps. Condition 4 assures that no finite subcomputation of a computation is considered a computation; thus infinite computations must be carried out forever.

Because of this nondeterminism, one may view an NDISM M as giving rise to a tree of potential computations from each initial snapshot, as Figure 1 illustrates. (All figures and tables are gathered at the end of this document starting on page 100.) The point in the far left represents an initial snapshot, and each forking point represents the choice of successor snapshots. A computation is a path from left to right along the tree. Note that the nondeterminism that represents concurrency is different from the traditional automata theoretic nondeterminism [Hopcroft79]. Automata theoretic nondeterminism is an abstraction of either (1) trying all possible computations at once until one is found that gives an answer to the problem that the automaton solves or (2) always choosing the right successor snapshot that leads directly to an answer. In concurrency, one is choosing only one path, and there is no notion that any one is more correct than the other.

A computation $C=(s_0, \ldots, s_{i-1}, s_i, \ldots)$ in an NDISM $M=(I, I_0, F)$ is said to halt at snapshot s_n if and only if

1. s_n is in C
2. $F(s_n)=\varnothing$

Note that Condition 4 of the definition of a computation guarantees the uniqueness of s_n; it is the first and only snapshot in C that can satisfy Condition 2 of the same definition.

The execution of a program p in the presence of input i is a computation beginning from an initial snapshot s_0, which contains some representation of p and i. For a given p and i, there may be more than one computation, some of which halt and some of which do not halt. If a computation halts, the computation is said to yield an output, namely a portion of the final snapshot designated as the output o. There are some special situations that have special names.

- If for all input i, all computations arising from i and a program p halt, then p is said to be a *halting program*.
- If for all input i, all computations arising from i and a program p do not halt, then p is said to be a *looping program*.
- If for all input i, the initial snapshot built from i and a program p gives rise to at most one computation, then p is said to be a *deterministic program*.

Note that a program that is not a halting program is not necessarily a looping program, as it may have some computations that do halt and some that do not halt.

A deterministic halting program, which is actually not the subject of this module, is said to implement a *function* because for each input, the program yields a unique final snapshot from which a unique output may be extracted. A deterministic non-halting program is said to implement a partial function; one assigns *undefined* as the result of the function for those inputs giving rise to non-halting computations. A nondeterministic halting program can also be considered a function, on the inputs to sets of all possible outputs for the inputs. A nondeterministic non-halting program can likewise be considered a function to sets some of which may contain *undefined* as elements corresponding to non-halting computations.

A looping program considered as a function is a very uninteresting function, as all of its results are *undefined* or singleton sets containing *undefined*. More usually, looping programs are called *systems*. Normally an operating system is supposed to be a looping program. In reality, operating systems are really non-halting, non-looping, nondeterministic programs. Their halting computations are considered erroneous! A challenge of program verification is to prove that programs that are supposed to implement functions do and that programs that are supposed to implement systems do.

In the usual formal model of concurrency, the snapshots contain some structure representing processes, each of which is in one of the three abstract statuses, awake, asleep, and terminated. When F is applied to a snapshot s_i, the resulting set of snapshots contains one element for each awake process in s_i. The result snapshot for a given process shows the changes caused by having that process execute one instruction. Thus choosing an element of $F(s_i)$ amounts to selecting one process in s_i to have it execute one instruction. The formal model then models the concurrency with an interleaving at the level of one instruction.

The question can be asked "Is this interleaving model an accurate depiction of concurrency that may include genuine parallelism?" To see that the answer is "Yes!" observe that if the processes shared no data objects (communications channels included) then no process can affect another, and the results of a computation involving these (independent) processes is indistinguishable from those of any, even serial, ordering of the processes. The only way processes can affect each other is through shared data objects. As an example of access to shared data, consider now two processes, being run by two processors, trying to write to the same memory location. In all machines known to this author, there is some hardware *arbiter* that prevents two processors from writing to the same memory location at the same time and serializes these writes so that one is finished entirely before the other begins. The net effect will be the effect of the second write. This arbiting occurs on every device and guarantees that every possible interaction of otherwise independent processes is serialized to the level of the machine instruction. Thus, any possible concurrent, even parallel, behavior can be simulated by an interleaving model that interleaves at the instruction level. This unit of interleaving is called the *granularity* of the interleaving. Thus, it is legitimate to use an interleaving formal model to represent concurrency.

II. Properties of Concurrent Programs

Recall the discussion about safety and liveness properties in [Bustard90]. Here, this discussion is recast in terms of the NDISM model. Later, when other formulations of these properties are given in other formal models, these formulations will be seen as expressing the same ideas.

Any property that one may wish to verify about concurrent programs may be categorized according to the nature of the property involved. The property may usually be classified as a *safety* property or a *liveness* property. The difference between them, as described in detail below, is in the nature of the quantifiers used over snapshots in their formal expression. This classification captures most of what is desired to prove. There are other properties that cannot be classified as either safety or liveness properties.

For each property described below, enough of a formal statement is given to show why it is a property of the type claimed. The only quantification shown explicitly is that over snapshots. The assertions in the scopes of these quantifiers are given in English. These assertions may obviously contain implicit quantifiers, but none of them are over snapshots; rather they are over elements within a snapshot. Such quantification does not change the nature of the property.

All the kinds of properties are described in terms of the NDISM model introduced earlier.

1. Safety Properties

A safety property is one that can be expressed in an assertion involving universal quantification over snapshots. Examples, in the abstract, of such properties are those that can be expressed thusly

- P is true in all snapshots, i.e., $\forall s \text{ in } C\,(P(s))$
- P never happens, i.e., $\forall s \text{ in } C\,(\neg P(s))$
- Whenever P happens Q is true, i.e., $\forall s \text{ in } C\,(P(s) \supset Q(s))$

Note that the C mentioned is implicitly universally quantified over all computations of the program about which the property is being proved. The program together with any set of input files determines an initial snapshot which, in turn, determines a set of computations of that initial snapshot.

Historically these properties are quite familiar, because they can be demonstrated by inductive means [King80, Cousot82, Manna72]. That is, they are shown to hold for initial snapshots and then they are shown to be preserved by the transition function F of the NDISM. They are very often expressed without the quantifier, relying on implicit universal quantification of free variables.

a. Deadlock Freeness

In no snapshot are all processes asleep or terminated, i.e., $\forall s \text{ in } C$, there is at least one awake process in s.

b. Mutual Exclusion

In no snapshot are two or more processes in their critical region, i.e., $\forall s \text{ in } C$, if one process in s is in its critical region then no other process in s is in its critical region.

c. Data Security

In no snapshot are there unauthorized disclosures, i.e., $\forall s$ in C, if a process in s is writing data to a file in s, then the data it is writing is from an object in s to which the process has access.

d. Proper Termination

If and when the computation halts, then all processes are terminated (i.e., none are asleep), i.e., $\forall s$ in C, if $F(s)=\varnothing$ then no process in s is asleep. Note that if $F(s)=\varnothing$, then no process in s can possibly be awake. Were there an awake process in s, it would be selected for execution to yield a next snapshot.

e. Partial Correctness

If at the beginning of the computation, the input is legitimate, and if the computation halts, then the output is what is required. In other words, if at the beginning of the computation, the input is legitimate, then in any final snapshot, the output is as required. In other words, if the input of s_0 in C satisfies input condition I and $\forall s_i$ in C, if $F(s_i)=\varnothing$, then the output of s_i satisfies output condition O.

Alternatively if at the beginning of the computation the input is in the range of the function to be implemented and if the computation halts, then the output is the result of applying the function to the input, i.e., if the input i of s_0 in C is in the domain of the function f, and $\forall s_j$ in C, if $F(s_j)=\varnothing$, then the output o of s_j is such that $o \in f(i)$.

Note that the second definition accommodates nondeterministic functions.

2. Liveness Properties

A liveness property is one that can be expressed in an assertion involving existential quantification over snapshots. Examples of such properties are those that can be expressed as the following:

- There exists a snapshot in which P is true, i.e., $\exists s$ in $C\,(P(s))$
- There exists a snapshot in which P is not true, i.e., $\exists s$ in $C\,(\neg P(s))$
- Whenever P happens then at some later time Q is true, i.e., $\forall s_i$ in $C\,(P(s) \supset (\exists s_j$ in $C\,(j>i \wedge Q(s_j))))$

The difficulty with liveness properties is that they cannot be demonstrated directly by the familiar inductive methods. They must be demonstrated by showing that each step in the computation brings the computation closer to a snapshot in which the desired property is true. One typically finds a well ordering (that is, a partial ordering with a least element) [Manna74] and program variables, or a function thereof, such that the variable or function values are related by that well ordering such that

1. within every n steps, with n fixed, the variables decrease with respect to that ordering, and
2. that the variable has reached the minimum implies that the desired property holds

Since a well ordered set has a least element, it is inevitable that the computation will eventually reach a snapshot with the desired property. One typically shows that the variables decrease by appeal to some safety, i.e., invariant, property. There is the need for considerable creativity in finding variables, some function of them, and a relation over the range of that function that allow one to carry out the proof. It should be clear from the complexity of the above description that showing liveness is considered much harder than showing safety.

Because proofs of liveness properties usually require showing some invariant properties along the way, it is no problem if the existential quantifier is within the scope of a universal quantifier.

a. Fairness

In all snapshots for all ready processes, eventually the process will become running; i.e., $\forall s_i$ in C, for all Π, a ready process in s, $\exists s_j$ in C, such that $j \geq i$ and Π is running in s_j.

b. Receiving Requested Data

In all snapshots, if a request for information has been received, then there is a future snapshot in which this information, if legitimate to do so, is released, i.e., $\forall s_i$ in C, if Π in s_i has requested data from an object o in s_i and Π has access to the object o, then $\exists s_j$ in C, such that $j \geq i$ and Π in s_j receives the data.

c. Sent Message Will Arrive

In all snapshots, if a message is sent, then in some future snapshot, the message will be received, i.e., $\forall s_i$ in C, if a message m is sent in s_i then $\exists s_j$ in C, such that $j \geq i$ and m is received in s_j.

d. Each Request Serviced

In all snapshots, if a service is requested, then there is a future snapshot in which this request is granted, i.e., $\forall s_i$ in C, a service is requested in s_i, then $\exists s_j$ in C, such that $j \geq i$ and the request is granted in s_j.

e. Termination

Eventually the computation halts, i.e., $\exists s_i$ in C, such that $F(s_i)=\varnothing$

f. Total Correctness

If at the beginning of the computation the input is

legitimate, then eventually the computation halts and the output is what is required, i.e., if the input of s_0 in C satisfies input condition I, then $\exists s_i$ in C, such that $F(s_i)=\varnothing$ and the output of s_i satisfies output condition O.

Alternatively if at the beginning of the computation the input is in the range of the function to be implemented, then the computation halts and the output is the result of applying the function to the input, i.e., if the input i of s_0 in C is in the domain of the function f, then $\exists s_j$ in C, such that $F(s_j)=\varnothing$ and the output o of s_j is such that $o \in f(i)$.

3. Others

These properties cannot be described either as universal quantification over unquantified snapshot assertions or as possibly universally quantified existential quantification over snapshot predicates. A non-exhaustive list follows:

- P happens infinitely often — for all snapshots, there exists a later snapshot such that P, i.e., $\forall s_i$ in $C(\exists s_j$ in $C(j>i \wedge P(s_j)))$.
- Whenever P happens, within n steps, Q happens, i.e., $\forall s_i$ in $C(P(s_i) \supset \exists s_j$ in $C(j \geq i+n \wedge Q(s_j)))$.
- Whenever P happens, within x seconds, Q happens, that is, $\forall s_i$ in $C(P(s_i) \supset \exists s_j$ in $C(|Time(s_j)-Time(s_i)| \leq x \wedge Q(s_j)))$.
- The transition satisfies constraint P [Locasso80, Scheid86a] — every transition satisfies P, i.e., $\forall s_i, s_{i+1}$ in $C(P(s_i, s_j))$

III. Formal Models

This section describes several well-known formal models of concurrency. Each of these is intended for and has been used for formal verification of properties of concurrent programs. A later section details the proofs that have been done with each. In this section, each is described in terms of its relation to the NDISM model described in Section I. Two considerations dictated the choice of the NDISM as the model of Section I.

1. The NDISM, which turns out to be an operational model, is one of the most fundamental in that it can capture all known phenomena, it corresponds to what is implementable and thus what will be found in implementations, and it can be used to model all the others.
2. In fact, models similar to NDISMs *are* used by other authors to describe their formal systems.

For each formal model, the description includes definitions of the basics and a brief discussion of what can be proved using it.

1. Operational — NDISM

Operational models attempt to specify the system behaviorally, that is by describing what happens during its computations. In every operational model is lurking an NDISM of some form. That is, in every operational model one will find a description of the snapshots or states, a description of initial snapshots, and a description of how to obtain a next snapshot from the current one. Some operational models are described with programs and others are described by other formal means.

a. Description of NDISM by Program

When one uses a program to describe an operational model, one is in effect writing an interpreter program which computes from initial snapshots. The snapshots are described by the declarations of the data structures needed by the interpreter; the initial snapshots are described implicitly as that configuration of the snapshot data structures that are accepted by the interpreter as being legitimate to begin a computation. The transformation is defined implicitly as what the interpreter does in modifying one snapshot into the next. Usually, in the interpreter there is a single identifiable point at which the configuration of the snapshot data structures are considered to form a new snapshot; in a normal interpreter this would usually be at the top of the loop in which instructions are fetched and executed. Examples of these kinds of definitions are the original and several later LISP definitions [McCarthy65] [Reynolds72] and the definition of Euler [Wirth66]. Such definitions are in general hard to use for verification. The transformation function is only implicit. Thus, one is forced to use the formal methods of verifying normal, sequential program behavior [Berztiss88] just to show that the interpreter is doing what it is claimed to, so that one can show that the interpreted program is doing what it is claimed to. An NDISM in which the transformation function is expressed more directly is preferable.

b. Formal Mathematical Descriptions of NDISMs

The other languages for writing NDISMs have more mathematical notations for expressing the transformation in a manner that allows a more direct use of parts of its definition in proofs. These languages provide some notation for describing the set of all snapshots and the set of initial snapshots as a subset of the set of all snapshots. In these languages, snapshots have been described as trees with labeled nodes and/or edges, lists, ordered tuples, sets, functions, etc.

There are a number of different ways of specifying the F of a particular NDISM. These ways may be classified by a number of different dimensions. There are direct methods in which F itself

is described and there are indirect methods in which a program P which computes F is described. A program P *computes* F if and only if each computation of P is a computation induced by F and vice versa.

(i) Direct Description of F of NDISM

Within the former methods, there are two different approaches to directly describing F:

- Write a function which given a snapshot yields the set of next snapshots, and
- write a predicate satisfied by legitimate pairs of successive snapshots.

In these approaches of direct specification of F, it is easy to see the individual snapshot transitions but hard to visualize a program P computing the transitions. It may, however, be explicitly stated along with the NDISM that the transition F is not intended to represent primitive or indivisible transitions of the specified system. In this case, one cannot know the individual primitive snapshot transitions. Some examples of languages of the first kind are VDL [Lucas69, Wegner72], and SPECIAL [Silverberg79]. Some examples of languages of the second kind are Ina Jo™[3] [Locasso80, Scheid86a], and AFFIRM [Thompson81].

(ii) Indirect Description of F of NDISM

Within the latter method, any programming language, either of the implemented or gedanken variety can be used. In the gedanken variety, e.g., the language of the Euler definition [Wirth66], one finds languages with sets as data types and not-so-easily-implemented set theoretic operations, including quantification, as operations. In these methods of writing a program P, the program computing F is explicit, but it is hard to see the individual transitions. Indeed, it is usually left unspecified as to what are the indivisible operations in order to give the compiler the right to choose the primitive operations. For example, for

```
x:=x+1
```

the indivisible steps depend on the machine for which the code is generated. Usually the code is something like

```
LD      1,x
AD      1,=1
ST      1,x ,
```

resulting in three indivisible steps for the assignment. On the other hand, if the machine has a special instruction for incrementing a memory location by one, the statement might indeed be compiled into a single indivisible step,

```
+1      x .
```

Lamport [Lamport80a] has suggested a way to indicate indivisible operations at the source language level by use of angle brackets, "<, >", around indivisible steps. A decomposition equivalent to the first translation above would be specified as

```
<x:=<<x>+1>> ,
```

while a decomposition equivalent to the second translation would be specified as

```
<x:=x+1> .
```

c. Redundant Specification of Properties

In all of these methods of specifying F, it is sometimes allowed to specify non-algorithmic properties as a redundant description of what the computation is supposed to be doing. For example, one may have specified in F a database system in which each transition represents an update of or a query to the database. The redundant description might describe integrity properties that the data of the database always satisfy. The purpose of this redundant specification is to allow checking or even verification that the operations as specified preserve these properties.

Redundant Properties

In the direct methods of specifying F, one might give assertions that are to be satisfied by initial snapshots, by all snapshots, and by final snapshots. In the indirect methods, one might give a number of program point-assertion pairs such that every time execution goes through a point, its assertion is to hold. For example, consider a program to sort the elements of an array into ascending order. Attached to the entry point of the outermost loop, which steps through the array in ascending order, might be an assertion claiming that the portion of the array between the beginning and the element indexed by one less than the current loop index value are sorted and are less than all elements in the rest of the array.

Many of the methods of verifying what systems or programs do consists in verifying that the F or P are consistent with their redundant descriptions. Typically for the direct methods of specifying F, an inductive approach is taken to prove that a specified property holds in all snapshots. It is shown that the property holds in all initial snapshots, and then it is shown that *if* the property holds in any snapshot S, *then* it holds in any snapshot yielded by applying F to S.

[3] Ina Jo is a trademark of Unisys Corporation.

Safety Properties

In principle any expressible property can be proved in these operational models. However, because of the inductive nature of the typical proof, they have been used primarily for safety properties. It is quite straightforward to express a safety property as an assertion that must be true in every snapshots. One proves the property inductively by showing that all initial snapshots satisfy the assertion and then showing that if it holds at any snapshot, it will hold again at any next snapshot. Showing the latter amounts to showing that the transformation preserves the holding of the assertion.

Liveness Properties

Generally, liveness properties are not specified and proved. In some cases, the formal basis is not even in the model. For example, in the Ina Jo language, there is explicitly no requirement that any transform (the transformation of the NDISM is the disjunction of the transforms) be done. All one is allowed to show is what will happen *if* a transform is done. Moreover, at any snapshot, there is no guarantee that *any* transform will be applied. In any case, even if the necessary assumptions are present, liveness properties require existential quantification over snapshots. This is considered more difficult than plain universal quantification. Indeed if one has *only* universal quantification over snapshots, one is not obliged to work with any quantification over snapshots; the properties that hold on these universally quantified snapshots are proved invariant by inductive means. In fact in the cases of the Ina Jo language, AFFIRM, and SPECIAL, the accompanying interactive or semi-automatic proof system simply cannot handle quantification over snapshots; and there is no way to express them in the language of the system.

Operational models can handle both halting and looping programs. Unlike a functional treatment which assigns *undefined* to each non-halting computation, operational models consider the sequence of snapshots in a computation as the meaning of any initial snapshot. Hence non-halting computations are distinguishable by the individual infinite computation sequences.

d. Formal Verification of Redundantly Specified Properties

There are a number of approaches to verifying properties of NDISMs. The approaches depend on how the F of the NDISM is specified, either directly or indirectly.

(i) For Directly Specified F of NDISM

If F is specified directly, the usual interests are either of the following:

- Verify certain properties hold about the NDISM specified by F, or
- verify that an implementation of the NDISM is correct.

If one is to verify a property of the NDISM itself, one states the property in the language of the NDISM, say as an assertion involving elements of the snapshot or as a property that F must satisfy or a combination of both. Then the assertion is proved by induction over the length of the computation.

If one is to prove that an implementation of the NDISM is correct, then one must first specify the implementation either as another NDISM or as a program P implementing the computation induced by F.

If the implementation of an NDISM M_d is given as another NDISM M_g,[4] then the proof is carried out by showing that M_g simulates M_d by a proof technique [McGowan71, Berry72] that is similar to that used in automata theory work.

One shows that for each computation C_d in M_d, there exists a computation C_g in M_g which behaves the same way. By behaving the same way is meant that there is a function $\psi: I_g \rightarrow I_d$ which builds each snapshot of C_d from its corresponding snapshot in C_g. That ψ has this property is shown by mathematical induction on the length of the computation C_d.

Many times, C_g simulates C_d in a lock-step snapshot-by-snapshot fashion. However, many times, this lock-step simulation is not possible; it takes several steps in C_g to implement a single step in C_d. The meaning of correspondence can be changed to allow ψ to build only some of the snapshots of C_d out of only some of the snapshots of C_g. The constraint is that a gap of no more than some fixed number is allowed between successive building snapshots in C_g and between successive built snapshots in C_d.

The lock-step situation is illustrated in Figure 2. The *Support Materials for Concepts of Concurrent Programming* has a full formal definition of simulation of one NDISM by another.

[4] The subscript "d" signifies implementeD, and the subscript "g" signifies implementinG.

If on the other hand, the implementation of the NDISM is not another NDISM, but is in fact code in some programming language, then two possibilities exist:

- The whole NDISM is implemented by a single program *P*. The correctness issue is whether *P* implements the whole of *F* down to the nondeterministic choices and the potential unbounded computation length.
- The *F* can be decomposed into a collection of individual operations, each of which does one of the possible transformations. The operation of *F* consists, in each application, of selecting one of these transformations and executing only it. In this case, the usual implementation is built of a collection of procedures for the individual operations and of a previously cooked shell which chooses the right operation at each step and invokes the corresponding procedure. The correctness issue is then whether the individual procedures implement the individual operations, the shell being presumed correct.

In either case, one ends up using the various program verification techniques, such as that of Hoare [Hoare69] in which one shows an actual program to satisfy certain properties [Berztiss88].

(1) NDISM Implemented by Single Program

In the first possibility, if the NDISM truly exhibits non-deterministic behavior and the non-determinism is part of the specified behavior, e g., to specify concurrency as opposed to possible allowed non-concurrent computations, then the program will have to exhibit the same non-determinism in the form of concurrency. The NDISM specification will have to be converted somehow into a specification that is used by one of the various methods to prove program behavior. If the NDISM halts for all desirable computations, then input, output, and most likely invariant assertions will have to be generated to describe the program's behavior. If the NDISM intentionally does not halt for all desirable computations, then output assertions are useless, as they vacuously hold. Moreover, invariant and so-called "eventually" assertions, i.e., liveness assertions, will have to be generated to describe the program's behavior. Then the chosen proof method is used to verify that the program has the desired behavior. If on the other hand the NDISM's non-determinism is merely to state allowable computational orders, any allowed order is considered correct, and it is not required to be able to do them all, then standard methods for non-concurrent programs are to be used. These are outside the scope of this module and the reader should consult [Berztiss88] for more details.

(2) NDISM Implemented by a Collection of Invokable Procedures

In the second possibility, the problem has been reduced to showing the correctness of a collection of non-concurrent subroutines implementing non-concurrent transitions, whose only non-determinism is for the purpose of allowing any one of several possible results. The concurrency has been pushed down into the process of selecting which of the available transitions will be invoked next. If one verifies that the procedures do implement the transitions, then to the extent that the basic invoking shell works, the collection of procedures plus the shell implements the NDISM. The meaty part of this verification has been reduced to the standard non-concurrent variety which is outside the scope of this module.

(ii) For Indirectly Specified F of NDISM

If *F* is specified indirectly with a program *P*, the interests are the same as for a directly specified *F*, i.e.,

- to verify that an implementation of the NDISM is correct, or
- to verify certain properties hold about the NDISM specified by *P*.

It is rare that one has to prove the correctness of an implementation of an NDISM specified by a program, particularly when the specifying program is executable. However, when it is done, particularly when the specifying program is not executable, it is done by proving the implementing and the defining programs equivalent. This proof can be done by showing that

- both compute the same function,
- both satisfy the same formal specifications, albeit input/output or temporal logical,
- a series of correctness preserving transformations [Balzer81, Balzer85] modifies one (usually the defining one) into the other, or
- one is compiled into the other.

Verifying that certain properties hold involves specifying these properties in the form of assertions which are attached to the program as a whole or to various places in the program, depending on the method being used. Then the code is proved consistent with the assertions by

the method being used. For example, if one is using Hoare logic, the assertions will be sprinkled around the program, while if one is using temporal logic, the assertions will be attached to the program as a whole. Assertions attached to the program as a whole describe properties of the program's computation as a whole in terms of particular snapshots that exist at various times during the computation. Assertions attached to points in the program describe the snapshot at any time execution passes through that point.

e. Graph Models of Concurrent Computation

The various graph-based systems such as control- and data-flow schemes, Petri nets [Peterson81], etc. are also operational. However, the language for expressing the parts of the NDISM is pictorial. Associated with each is a description of how a diagram is to be interpreted as specifying a computation. Usually this description says something to the effect of

> Select some process node, all of whose input arcs contain tokens. Fire that node, remove one token from each input arc, and deposit one token on each output arc.

That description may be stated in natural language, some programming language, some mathematical language or some mixture of all of these. Generally what the processes in the nodes do is left unspecified or, to use the terminology, of the models *uninterpreted*. Those that model data flow as well as control flow state which data objects are used, but in the uninterpreted domain, they say nothing about the actual values of the data. Because there are no interpretations as to what the processes do, the set of computations that can be described is limited. One uses such models strictly to focus on the concurrency and synchronization issues. The philosophy behind these uninterpreted models is that if a property can be proved of an uninterpreted model then it holds for all interpretations of that model. One might be able to prove mutual exclusion for a program just on the basis of its control structures, independent of any values of any objects. Because of the limited functionality in the models, it is generally easier to carry out proofs; there are fewer operations to consider. On the other hand, this lack of functionality may prevent proving properties that hold only because of the values of objects and not strictly because of the structure of the programs.

Because of the limited functionality, not all properties expressible in other operational models are expressible in these pictorial models. For example, it is impossible to demonstrate partial or total correctness without knowing the values of objects. However, one can express proper termination as termination, i.e., having no fireable node (no node has tokens on all of its input arcs) while having less than some fixed number of tokens on each arc. For such control flow schemes, proper termination is even decidable.

2. Axiomatic

Axiomatic systems for dealing with concurrency provide three main ingredients,

1. an assertion language for describing snapshots at arbitrary points in a program's execution, usually the language of first order predicate calculus with equality,
2. a set of axioms for describing the behavior of primitive statements,
3. rules of inference for combining behaviors of constituent statements into behaviors of the containing constructs such as loops, conditionals, programs, etc.

Hoare Logic and its Limitations

Axiomatic systems for dealing with concurrency are generally based on Hoare's logic [Hoare69] or Dijkstra's weakest precondition logic [Dijkstra76]. The main difficulty in using these logics directly for dealing with concurrent programs is that because of the potential of interference, their axioms do not work. For example, the axiom for assignment is

$$\{P_e^x\}\ x:=e\ \{P\}$$

which when adapted to the assignment statement

$x := x+1$

under the condition that

$$x=1$$

prior to the assignment yields

$$\{x=1\}\ x:=x+1\ \{x=2\}\ .$$

However, as mentioned in Section II.3.d, when this assignment is executed in the presence of other interfering processes, the value used to compute $x+1$ may not be the same as is mentioned in the input assertion.

The various extensions to the axiomatic logics deal with this interference problem in different ways. The axiomatic systems surveyed are

- the Owicki-Gries extension of Hoare logic,
- the Lamport extension of Hoare logic, and
- the Lamport extension of Dijkstra's weakest precondition logic.

Owicki-Gries's Extension of Hoare Logic

Owicki and Gries [Owicki76a, Owicki76b] use ordinary Hoare logic axioms but put non-interference

requirements into the antecedents of rules of inference. For example, the rule they give for the parallel-execution statement

resource r_1*(variable list)*, ... , r_m*(variable list)*
cobegin S_1 || ... || S_n **coend**

is as follows.

If $\{P_1\}S_1\{Q_1\}$ and ... $\{P_n\}S_n\{Q_n\}$ and no variable free in P_i or Q_i is changed in S_j with $i \neq j$, and all variables in $I(r)$ belong to resource r, then

$\{P_1 \wedge \ldots \wedge P_n \wedge I(r)\}$
resource r_1*(variable list)*, ... , r_m*(variable list)*
cobegin S_1 || ... || S_n **coend**
$\{Q_1 \wedge \ldots \wedge Q_n \wedge I(r)\}$.

This rule says that in the case of parallel execution of statements, the normal axioms and rules may be used for the individual statements if there is no chance of interference between them. The rule puts upon the prover the obligation to show non-interference.

Lamport's Extension of Hoare Logic

Lamport takes sort of an opposite approach, of changing the meaning of the assertions and the axioms for primitive statements so that they catch interference in a slightly different way. In the Lamport logic,

$$\{P\}\,S\,\{Q\}$$

means, "if execution is begun *anywhere in S* with the predicate P true, then executing S will leave P true while control is inside S, and will make Q true if and when S terminates." Thus, Lamport's treatment of the assignment

x:=x+1

under the condition that

x=1

prior to the assignment is

$\{x=1 \wedge [after('<x+1>') \supset value('<x+1>')=2]\}$
<x>:=<x+1> {x=2}

In this, angle brackets are used to surround operations that are atomic, i.e., cannot be interrupted and are guaranteed to be done within a single NDISM step both in the abstraction and in *any* implementation. The rule says that if x starts off as 1, and after doing x+1 the value of that subexpression, x+1, is still 2, then the assignment causes x to get 2. In other words, *if* there was no interference with x during the assignment, it behaves as an assignment in the normal nonconcurrent situation. Of course, it is on the prover to demonstrate that there is no interference. In this sense, this approach is equivalent to Owicki's.

Because these axiomatic methods prove that assertions which are attached to specific places in the program are invariant, they really address only safety properties.

Dijkstra's Weakest Precondition Logic

Dijkstra introduced a variation of the Hoare axiomatic system when he introduced a proof-directed discipline of programming [Dijkstra76]. The approach, that of *weakest precondition*, attempts to find the weakest conditions under which a given statement is guaranteed to halt and to yield a snapshot satisfying a given postcondition. For example,

$$wp(x:=x+1, x=2)$$

is

$$x=1\ .$$

because only when x=1 is x:=x+1 guaranteed to halt and yield x=2. The general rule for the assignment statement is

$$wp(x:=e, P) \equiv P_e^x\ ,$$

the obvious correspondent to the Hoare rule,

$$\{P_e^x\}\,x:=e\,\{P\}\ .$$

There is a dual for the weakest precondition called the *strongest postcondition*. The strongest postcondition, $sp(S, P)$, of a statement S and a precondition P is the strongest condition describing any snapshot yielded by the execution of S from any snapshot satisfying P.

The Hoare rule is a partial correctness rule, as its meaning carries no requirement that the statement halts. It provides only that *if* the statement halts *then* the postcondition is true; if the statement does not halt, then any postcondition is accepted as vacuously true. The weakest precondition formulation, on the other hand, finds preconditions which guarantee halting as well.

Lamport's Extension of Weakest Precondition Logic

Recall that most concurrent programs are systems that are supposed never to halt. For such programs, the weakest precondition would be false for any postcondition. Therefore, Lamport has developed a variant of the weakest precondition approach that is more useful for concurrent non-halting systems. For concurrent programs that do not halt, the interest would be to prove some invariant property, i.e., a safety property. Accordingly, Lamport [Lamport87] defines the concepts of weakest invariant, *win*, and strongest invariant, *sin*, and gives rules for deriving them for programs given the *wp*, *sp*, *win*, and *sin* for constituent statements.

I is an invariant of S if

$$\{I\}\, S\, \{I\}\ .$$

Lamport then defines S as leaving an assertion I *invariant* if and only if

$$I \supset wp(S, I)\ ;$$

it is also the case that

$$sp(S, I) \supset I\ .$$

Then, $win(S,Q)$ is the disjunction of all predicates I such that $I \supset Q$ and S leaves I invariant, and

$sin(S,P)$ is the conjunction of all invariants I of S such that $P \supset I$.

Lamport is able to extend the Owicki-Gries method to be able to reason about programs for which the atomic operations are not specified. However, note that the weakest and strongest invariants do not give the power to work with liveness properties.

3. Temporal Logic

Temporal logic [Pnueli77, Pnueli81] is an attempt to deal with the logic of computation sequences without succumbing to the difficulties of quantification. One is able to talk directly about sequences of snapshots, all snapshots, and the existence of a snapshot that have desired properties. The direct expression is based on an axiomatized logic of time that obviates the necessity to quantify over snapshots.

In temporal logic, one is provided temporal operators, "henceforth" ($\Box$) and "eventually" ($\Diamond$) which can be applied to the standard assertions about snapshots. While they can be defined relative to each other axiomatically, any model of them makes use of some underlying NDISM. Hence, they are described here in that manner.

Basic Temporal Operators

Temporal semantics talks about assertions that hold true in the current snapshot or all or some future snapshots. Therefore, it will be necessary to identify which snapshot in a computation is the current one. The current snapshot will be called s_c in the following discussions.

$$\Box P \equiv \forall s_i (i \geq c \supset P(s_c))$$

$$\Diamond P \equiv \exists s_i (i \geq c \wedge P(s_c))$$

Note that occurring henceforth includes occurring in the current snapshot. That something eventually occurs includes the possibility of it occurring in the current snapshot.

Just as the universal quantifier and the existential quantifier are duals of each other so are $\Box$ and $\Diamond$.

$$\Box P \equiv \neg \Diamond \neg P$$
$$\Diamond P \equiv \neg \Box \neg P$$

It is useful to define some other operators that correspond to often used temporal relations. One such useful relation is "leads to" ($\longrightarrow$).

$$A \longrightarrow B \equiv A \supset \Diamond B$$

With these temporal operators it is easy to express both safety properties and liveness properties. To do these, it is necessary to let *INIT* be an assertion that is true at all and only proper initial snapshots; that assertion is needed to anchor the time to the start of the computation rather than to any arbitrary current snapshot.

Expressing Properties with Temporal Operators

Recall the general safety properties introduced in Section II.4.a.i.

- P is true in all snapshots, i.e., $INIT \supset \Box P$
- P never happens, i.e., $INIT \supset \Box \neg P$
- Whenever P happens Q is true, i.e., $INIT \supset \Box (P \supset Q)$

Recall also the general liveness properties introduced in Section II.4.a.ii.

- There exists a snapshot in which P is true, i.e., $INIT \supset \Diamond P$
- There exists a snapshot in which P is not true, i.e., $INIT \supset \Diamond \neg P$
- Whenever P happens then at some later time Q is true, i.e., $INIT \supset \Box (P \longrightarrow Q)$

Temporal logic can even capture some properties that are neither safety nor liveness, such as the concept of P happening infinitely often.

$$INIT \supset \Box \Diamond P$$

Provided with the temporal logic is a system of axioms and rules of inference for reasoning directly in the temporal domain without having to fall back on a model. In order to be able to prove things about a program using temporal logic it is necessary to provide a temporal logic-based semantics of one's programming language. Besides saying what each kind of statement does, the semantics provides a basic liveness property for each primitive statement that says that it eventually finishes. From this liveness property and the semantics of the various constructs, the prover is able to prove liveness properties of whole programs.

Linear and Branching Time

There are, in fact, two alternative models of time in temporal logic, *linear time* and *branching time*. Both are intended to be used with nondeterministic computations. Recall in Section I, that two kinds of nondeterminism were identified, the kind that models concurrency and the kind that is used for automata-theoretic investigations of algorithms. Linear time corresponds to the concurrency-modeling nondeterminism, and branching time corresponds to the automata-theoretic nondeterminism. The temporal logic described above is in fact linear time, as it was stated that

$$\Diamond P \equiv \neg \Box \neg P$$

That is, if it is not true that P never occurs, it eventually occurs. This statement is true only if there is only one possible future, namely a single path down the tree of computations. "Not occurring never" means that it must eventually occur. However, in branching time, automata theoretic nondeterminism, the fact that it is not true that P never occurs means that there exists some computation in which P does occur; it may not occur in *all* possible futures, i.e., all possible computations from now. The formal treatment of branching time is to distinguish between the "eventually" and the "not never" operators. The former has the meaning implied by the operational model and the latter is the dual of the "henceforth" operator. However, since branching time is not really a good model of concurrency, it will be discussed no more in this module.

Another Temporal Operator

There is a third temporal operator used by some authors, e.g., Ben-Ari, Manna, and Pnueli [Ben-Ari81] and Hailpern [Hailpern82], namely the "next" ($\circ$) operator. It applies its argument predicate to the next snapshot.

$$\circ P \supset P(s_{c+1})$$

As the use of $\circ$ implies an explicit time scale or an explicit sequential progression of time, the tendency is not to use it. Using it would overspecify a computation to the point of saying that a certain event has to happen in the next snapshot and would not be acceptable if it were implemented as a multi-step procedure.

Temporal Logic Specification of Concurrency

Hailpern's Ph.D. thesis [Hailpern82] defines a useable version of temporal logic, describes a programming language, VALET, with processes and monitors, gives a temporal logic semantics for the language, and demonstrates how properties of programs in this language may be specified and verified. These ideas are then applied to specify and verify the correctness of programs implementing a number of network protocols and resource allocation systems.

In VALET, processes and monitors are written in the form of separate modules that may see and invoke each other. The form of a process module is similar to that of a Pascal program. It declares local variables and has a body consisting of a sequence of statements which may contain calls to procedures offered by the monitor modules. Generally, the body of a process module is an infinite loop. A monitor module declares local variables and a collection of procedures invokable from outside the monitor; in fact, in general, no module's local variables are visible from outside the module. The semantics of these modules is that all process modules are invoked to run concurrently at the beginning of the computation. All monitors are invoked also in order to have their initialization code executed. Then the monitors sit and wait until monitor procedures are called by the processes. The operational rule is that no more than one process can be executing the body of any procedure inside a given monitor. The at most one process that is executing inside a monitor procedure is said to *currently own* the monitor.

For a particular system involving process and monitor modules one gives specifications as follows for each module:

1. For each process, one gives an invariant and a commitment.
2. For each monitor as a whole, one gives an invariant.
3. For each monitor procedure, one gives a service specification and a commitment.

Invariants

Among the invariants above, there are two kinds which are somewhat different from the more familiar loop invariant, which is true each time control passes through the loop cut point and which may not be true at other points in the loop. A loop invariant would be specified as the following kind of safety property:

$$\Box(\textbf{at } CutPoint \supset I)$$

where I involves only variables visible at the cut point. The invariant of a process is a property of variables visible to the process which is true at *all* times. It would be specified more simply:

$$\Box(I)$$

where I involves only variables visible to the process. A *monitor invariant* is a property of the local variables of the monitor which is true at all times, except possibly during the execution of the body of

a monitor procedure. This exception allows monitor procedure bodies to make temporary changes that may invalidate the invariant so long as they restore the holding of the invariant upon exit. For example, in a monitor which keeps a linked list of the resources it is managing, the invariant would describe a well-formed linked list. The monitor procedure to remove an item from the list would temporarily invalidate the invariant as it performed the pointer manipulations to rebuild the list after removing the item. However, after the list is rebuilt, the invariant would once again hold. It is safe to let the monitor procedures invalidate the invariant temporarily because it is guaranteed that at most one process can be inside any procedure of a monitor at any given time. Therefore, it is not possible that any other process's concurrent operation can mess up the list while it is not in proper form. It is on the monitor writer to insure that all invariants still hold whenever a process can relinquish ownership of the monitor, namely upon a wait statement or upon return from a monitor procedure. It is this invariant that is called *I* in the rule for the monitor procedure body.

The invariant of a process is proved by standard safety assertion verification methods. That is, it is shown that the invariant holds upon initiation of the process and that each statement inside the process preserves its holding. The invariant of a monitor is demonstrated to hold after the end of its initialization part, and then each monitor procedure is demonstrated to preserve the invariant's holding across, but possibly not within, its body.

Service Specification

A service specification for a monitor procedure is simply its input/output specification, i.e., the *P* and *Q* of the rule for the monitor procedure body. This is demonstrated by standard safety assertion verification methods.

Commitment

A commitment is a liveness assertion, as it states properties that are guaranteed to happen. Note that a service specification does not really guarantee very much since it is only a statement of partial correctness, i.e., what happens *if* the body halts. A commitment, being a liveness assertion can carry a guarantee of something happening, especially if it contains an occurrence of the eventually operator. Indeed, it is for this reason that processes, which generally loop and do not halt, do not have service specifications. A process's commitment states the progress that it guarantees to have as it loops indefinitely.

The liveness assertions of a process are proved by composing the liveness assertions of its statements and the liveness assertions of the monitor procedures it calls. As mentioned, liveness assertions of the monitor procedure bodies are verified by composition of the liveness assertions of their constituent statements.

Advantage of Temporal Logic

The advantage of temporal logic in working with concurrency is that temporal logic expressions read more like the natural language temporal statements than do the equivalent snapshot quantified assertions. For example, the temporal logic expression

$$sent(M) \supset \# \Diamond \#(arrive(M))$$

captures the natural language statement

"If a message is sent then eventually it will arrive"

much more naturally than does

$$sent(M,s_c) \supset \exists s_i (i \geq c \wedge arrive(M,s_i))$$

Certainly, the temporal logic expression is more readable than is the quantified assertion. This more natural flavor of the temporal logic allows it to be more abstract concerning the passage of time. Even if the logic is that of branching, discrete time, one can write specifications that hide the detail of discrete passage of time and express that something happens just "eventually". With quantification over snapshots, one must show this aspect of the formal model, even though it is irrelevant to what is being specified.

Indeed, obtaining this ability to abstract away from discrete time passage was the rationale behind Nixon and Wing's proposal to add temporal logic to the Ina Jo specification language [Wing89]. The Ina Jo language is for writing operational specifications and in fact does *not* permit quantification over snapshots. It was proposed to add the temporal operators by defining them axiomatically in terms of each other rather than by reducing them to formulae with quantification over snapshots.

Two programming notations or program specification notations have been developed in the temporal logic mold. One is is a notation for specifying concurrent program modules, introduced by Lamport [Lamport83a], and the other is UNITY developed by Chandy and Misra [Chandy88]. Both use whatever is convenient from mathematics for expressing values of variables. Both allow concurrency at the level of the assigment statement. The meaning of a program is expressed in terms of assertions about states that hold at specific points or globally, either invariantly, occasionally, or at specific instances. There are a variety of temporal operators allowing expressing both safety and liveness properties. Lamport's description, being limited to a journal article, is not formally complete. The Chandy and Misra book gives full details on the language and describes the meanings of statements and the temporal operatos axiomatically. With the help of

Boyer-Moore logic, the UNITY proof system has been mechanized [Goldschlag90a, Goldschlag90b].

Another language incorporating temporal logic is COL [Karam91]. COL is a linear-time temporal-logic based specification language, which can be used to specify concurrent Ada programs. Associated with COL is TimeBench, a concurrent system design environment that includes a deadlock analyzer. TimeBench is an extension of the second Buhr's CAEDE visual Ada program design environment and uses its notations for expressing program structures.

4. Denotational

There are two main aspects to a denotational semantics of a language.

- The semantics are described in a syntax-directed manner, that is the meaning of a construct is built from only the meanings of its direct syntactic components.
- The meaning of a program is usually, but not always, taken as the function on input to output that it computes.

Actually the word "denotational" is used to refer to a definition which is entirely syntax directed as described above. In principle, any semantic domain can be used, e.g., the computation sequence given rise to by an initial snapshot. However, the tradition is to use the function that a program computes as its meaning. A program which does not halt at some inputs is given as its meaning a function that computes *undefined* for those inputs. This tradition of course, limits the applicability of denotational semantics to function programs. This particular kind of denotational semantics would not be useful for modeling looping programs such as operating systems.

Nonconcurrent Flow of Control

In order to be able to model **gotos** and other wild flows of control without needing other than direct syntactic components, the semantics of each language feature is given as a function of itself, the current snapshot information, and a *continuation*. The continuation of a construct is the function computed by the part of the computation that follows the completion of the construct, i.e., the *rest* of the computation.

This continuation turns out to be the value of a label, so that doing a **goto** means replacing the continuation that exists at the point of the **goto** by the one which is the value of the label.

Concurrency and Processes

The facts that processes do **gotos** and labels are continuations leads directly to the conclusion that a process should also be a continuation. However, here what function is computed by this continuation is not clear. The normal continuation computes the final snapshot, if it exists, as a function of the current snapshot and the changes caused by the current statment. Thus, the continuation corresponding to a process should compute all possible final snapshots as a function

- of the current snapshot,
- which of the processes is selected to execute next, and
- the changes that the selected process's next statement causes.

In normal denotational semantics, continuations are composed to build other continuations. That is, the continuation from now until the end of the current loop is composed with the continuation from the end of the loop until the end of the computation to get the continuation from now until the end of the computation. Recalling the tree of Figure 1, consider how this composition would occur. In Figure 3, the left hand contour delimits the part of the tree denoting one continuation, say to the end of the current **cobegin-coend** construct. The right hand contour denotes the continuation going from the end of the **cobegin-coend** construct to the end of the computation. Their composition is not the whole tree. Figure 4 shows a computation tree with no forks; in such a deterministic case, the composition of the two parts of the path *is* the whole computation, as is desired. The reason for this is that in composing one choice's continuation with the continuation containing the choice, one loses the ability to choose the unchosen choices. Thus, concurrent continuations cannot be usefully composed. Other means must be found to build continuations. See the literature [Plotkin76, Smyth78, Berry85, Schmidt86] for more details.

Limited Concurrency

The result of these difficulties is that most denotational semantics of concurrency have focussed on non-shared memory concurrency in which the only interaction between processes is via messages sent along a communication channel that enforces synchronized access [Francez80]. The restricted interaction greatly limits the number of choices that have to be re-interleaved. Indeed, if two processes are totally non-interfering there is no need to interleave them. [Plotkin76, Plotkin83]

5. Comparisons of the Coverage of the Approaches

The view of the world is three dimensional, and the dimensions are

1. the nature of the program,
2. the nature of the property to prove, and

3. the definitional and proof approach.

The choices for each dimension are:

1. the nature of the program:
   - functional, i.e., halting
   - looping
2. the nature of the property to prove:
   - safety
   - liveness
3. the definitional and proof approach:
   - axiomatic
   - denotational
   - operational
   - temporal

The issue is, "For any configuration of two dimensions, which choices of the third are available." Since three dimensional paper does not exist yet, it is necessary to show the world as three two-dimensional tables, as Figure 5 shows.

The first of these shows all the approaches that can be used for any given property that is to be proved about any kind of program. The choices are more limited for liveness properties and for looping programs. Liveness properties cannot be handled by strictly inductive invariance proofs, and looping programs cause partial correctness and functional approaches to be to give no useful information. The functional approaches are those that try to treat all programs as implementing functions, i.e., the axiomatic and the denotational approaches.

The second table shows all the properties that can be proved for any kind of program using any of the discussed approaches. This table shows that the functional approaches cannot be used with looping programs to prove any useful properties. It is clear that safety is more universally covered than is liveness.

Finally, the third table shows all the kinds of programs for which one can prove the various properties using the various approaches. In this table, it is clear that it is much harder to deal with looping programs than functional programs, and that axiomatic approaches do not help at all when trying to prove liveness.

The following summarizes what is possible for the approaches.

- Operational models can deal with either functional or looping programs and are always useful for demonstrating safety properties. They may be used to demonstrate liveness properties only if the language permits quantification over snapshots; most do not.
- Axiomatic models based on partial correctness deal only with functional programs, and since they work with invariants, they are useful only for demonstrating safety properties.
- Temporal semantics, not being tied to partial correctness, can deal with either functional or looping programs and are useful for demonstrating safety and liveness properties.
- Denotational semantics, as traditionally used, can deal only with functional programs and are useful for demonstrating safety and liveness properties but only of processes that do not share memory. When processes share memory, the amount of interaction possible, and which must be dealt with formally, explodes to the point of intractability.

IV. Actual Specifications of Software

In the literature, a number of the above formal systems have been used to specify concurrent programs and in some cases, properties have been proved of them. The primary examples have been in verifying security of operating system kernels, verifying integrity of databases, and verifying the correctness of protocols.

1. Operating System Security

The approach taken by the various workers in the security area is an operational one. The main problem is to design an operating system that enforces a certain security policy. It is recognized that to prove an entire operating system adhering to any but the most trivial property, e.g., *true*, is hopeless because of the sheer size of the code for an operating system. Fortunately, for a security policy, it is not necessary to prove that the entire operating system enforces it. Most operating systems are structured in such a manner that a small kernel gets the job of allocating objects to users and enforcing the policy. If the kernel properly does its job and the only way the rest of the system and any other program or procedure can get to the protected objects is by invoking the kernel operations, then it is not necessary to worry about what the rest of the system or the other programs or procedures do. Thus, in a properly structured kernel-based operating system, it suffices to verify that the kernel does the job right and to arrange that the rest of the system and no other program or procedure can directly access the protected objects. The latter can usually be verified by inspection that no objects are accessed directly or it can be enforced by the compiler's normal symbol table mechanism; that is, the protected objects are not even visible in the symbol table. Thus, the problem of verifying the security of a kernel-based system is reduced to that of verifying the security of the kernel itself.

It has been said that the only programs that can be verified are toy programs. The kernel-based approach [Popek75, Popek79] aims to make the only program that has to be verified, i.e., the kernel, small enough to be a toy!

Security Policy Identification

In the typical effort to design and implement a verifiably secure operating system, the effort begins with an identification of the security policy [Landwehr 81, Landwehr83], that is, what objects are going to be protected, what is the nature of this protection, and from whom they are going to be protected. The usual situation is that files are the objects being protected and they are being protected against inappropriate access by processes representing users. For example, a file may be given a classification, e.g., secret or top secret, based on the sensitivity of its contents. That file would be protected from being read by processes representing users whose clearance is not at least equivalent to the classification of the file, i.e., secret or top secret, respectively. The system might also allow blind appending of this file by any process whose clearance is at least equivalent to the classification of the file. The formalized security policy most often quoted is that of Bell and LaPadula [Bell73].

Decomposition of System

Once the elements of the policy are identified, it is necessary to decompose the system into two parts, the kernel and the rest. The intention is that the kernel will be concerned with at least everything that can affect or be affected by the security policy, i.e., the files, the processes, and the access rights and the operations governed by these rights. The kernel is designed so that, in the case of the example, the only way to create processes and files is by asking the kernel to do so and the only way that processes can gain access to files for reading and writing is by asking the kernel for permission to do so, and the only way that processes can actually read from or write to files is by having obtained permission from the kernel. This means that in the rest of the system and in any other program running on the system, these operations on files and processes can be done only by invoking kernel routines or by invoking routines that ultimately invoke kernel routines.

This particular decomposition of a system into a kernel and the rest makes sense from the software engineering point of view. A glance at the way operating systems are designed, not for the purpose of enforcing security, but for the purpose of making an easy to understand and easy to maintain system, shows the same basic decomposition being followed [Dijkstra68, Organick72, Ritchie74, Comer84, Bach86, Tanenbaum87].

Specification of Policy

The specification of the kernel includes a specification of security policy that it must satisfy. The usual approach is to specify an NDISM in which F is a function that non-deterministically invokes kernel operations. That is, at each transition some *one* kernel operation is selected for execution. The security policy is expressed as an invariant and other assertions that the operations must preserve and satisfy. For example, that in all snapshots no process has read access to any file that is at a classification higher than its clearance is an invariant expressing part of the example policy.

This particular formal model has two important simplifying assumptions.

1. The model does not necessarily insist that the only thing that can happen during a computation is what is specified. Thus, the model assumes that any thing else that might happen is simply not *security relevant*. This is reasonable if it is known that all accesses to the entities involved in the security policy are described completely in the model [Kemmerer82].
2. The execution of each kernel operation is indivisible, so that it is a correct model to interleave at the level of their invocation. This is reasonable since in most systems, such operations are implemented in a non-interruptible manner so that they are effectively indivisible.

Iterative Design Process

The designers begin an iterative process which halts only when the operations as specified provably meet the security requirements as specified. They attempt to prove the operations as specified meet the specified requirements. Each failure to do so leads to closer inspection of both the operation specifications and the security policy specifications. Usually this examination leads to identification of the reason that the proof fails, and this identification leads to changes to the operations, their specifications, the security policy, and its specification. After changes are made, a proof is again attempted. It is probably this close inspection and subsequent changes that are the most valuable aspects of the method, far more valuable than the proof *per se*.

Program Development

Once the kernel specification has been verified to adhere to the desired security policy, two particular program developments may proceed. One of these is of the software that uses the kernel. The two main constraints on any such software are that

1. it invoke only kernel operations when it wishes to do something which can be security relevant and
2. these invocations assume the specifications given in the NDISM for the kernel.

The other program development is that of the kernel itself. Assuming that the kernel is structured as a collection of subroutines that can be invoked from outside the kernel, i.e., a collection of supervisor

routines, it suffices to show that each subroutine correctly implements its corresponding transition specification. The specification of each such transition tends either to be deterministic or non-deterministic with the intent of specifying allowed variations and not concurrency. Therefore proving these subroutines correct can be done with traditional non-concurrent methods, as covered in [Berztiss88]. The important thing here is the concept that if the specification of an operation meets a given requirement and one has shown that a procedure correctly implements the specification, then one has also shown that the procedure meets the given requirement.

A fuller description of the technique is given in [Cheheyl81]. This paper describes the general technique and four different specification languages. A small example illustrates each one.

The famous Orange book [Klein83] describes requirements for systems to be accepted as secure by the Department of Defense. It specifies both the basic policy that must be implemented and the *methods* by which the system must be developed and verified to implement the policy. A system cannot receive so-called Orange-book certification *unless* it was developed and verified in a manner that inspires confidence in the claimed security. In other words, how the system is designed and implemented is at least as important, if not more, than what policy it implements — a recognition of the importance of having a systematic method for engineering the secure system. One system that has been certified A1 Secure under the Orange book is SCOMP [Benzel84].

Real-Life Use of Technique

The method has been applied a number of times at the level of verifying that the specified NDISM meets the security requirements. To this author's knowledge, the method has never been carried out to the point that the implementation meets the security requirements. While this failure is an indictment of claims to practicality of formal proof methods, in practice the failure is not that serious. It is generally agreed that the hard part of the implementation of a secure system is getting the specification of the kernel right. The kernel procedures are small enough and generally simple enough that getting their implementations right is far easier.

Limitations

The method has its limitations.

- The system can be no securer than the specification of security. That is, if the desired concept of security is not covered by the specification, then the system may not satisfy it, and it cannot be expected to observe it. For example, the specified security policy may state that a process cannot get read access to files with a classification higher than its clearance. However, proving this will not prevent a process from reading snapshot changes caused by another process that can legitimately read the file. If that process purposefully changes snapshot at an agreed upon rate so as to signal the bits of the content of the file it can read, then it will manage to transmit the contents of the file to other processes that are not supposed to have access to the file.
- Because of the sheer difficulty of carrying out proofs, what is specified tends to be only the barest minimum of security relevant properties. No attempt is made to prove that the transitions do what they are supposed to. There is no guarantee that an operation to read a file in fact causes copying of the contents of that file, only that the copying does not happen unless the security policy permits it. In some cases, the formal system used cannot even guarantee that an operation once invoked will even terminate. The formal system handles only safety properties and not progress properties. Note that an operation that does not give the data that is requested is quite secure, but it is not very useful.
- The security is as good as the underlying assumptions of correctly performing compilers, hardware, etc. The software has still kept its promise if as a result of a hardware failure, all top secret files get spilled out to the line printer that is in a public reading room.

Network Security

Several authors have dealt witht the related problem of secure transmission of messages over a non-secure network, i.e., of making sure that the intended recipient and *only* the intended recipient of a message see the message [Good82, Britton84].

2. Database Integrity

Given a database scheme, consisting of a collection of data types, e.g., for relations or inverted trees, etc., operations that can be performed on the data, and a statement of what it means for the data to have integrity, the problem is to insure that the database will maintain that integrity.

Run-Time Checking

The more usual situation is to do the checking at run time. However, this is expensive. Moreover, it is not clear what to do if the data are discovered not to have integrity; the operation that made the change, which is the primary cause of the lack of integrity, may have been done long ago.

Verification Approach

Another approach is to verify, at design time, that the database as specified has the required integrity.

This approach to database integrity verification is an operational one which is similar to that of operating system kernel security verification. One defines an NDISM to model the database; I defines the scheme, I_0 defines the initial configuration, F defines the operations. The integrity constraints are then expressed as an assertion on I. To prove that the database has integrity, it suffices to show that the integrity constraints assertion is an invariant. That is the integrity constraint holds for all elements of I_0 and it is preserved by each operation [Leveson83], [Paolini81].

Once the NDISM definition of the database has been verified to have integrity, any correct implementation of it also has integrity. Furthermore, any database application programmed entirely in terms of operations of the NDISM is also guaranteed to have integrity; this is because all accesses to the database are through operations that are guaranteed to preserve integrity.

The upshot of this approach is that no additional run-time integrity checks are needed. It is known that the operations are designed never to get the data into a snapshot in which they do not satisfy the integrity constraints.

Limitations

The method suffers from the same drawbacks as does verifying security, e.g, the real integrity of the database is only as good as the specification of what integrity means. If the database is a concurrent one, i.e., several processes may be accessing the data at a time, then the same approach works. All that is required, as is the case with operating system kernels, is that the operations be indivisible or atomic.

Making Operations Atomic

There are a number of approaches to making the implementation of the operations atomic. The brute force method is to physically prevent more than one process at a time from accessing the database. This approach is wasteful, however. In a truly large database, it is quite likely that the concurrent accesses are to independent parts of the database. In such a case, concurrent access is not harmful, and disallowing it would slow down the operation of the database intolerably; Consider an airline reservation system in which only one agent world-wide could access the database at a time. A more useful approach is to use a two-phase commit [Bernstein87] in which a process does all the calculations necessary to do an update unprotected, but then just before it is about to write the changes, it grabs uninterruptible, unique access to the data, checks that all data that contributed to the new values have not changed, and if not, then writes the changes and releases access to the data. If the process finds that at least one contributing datum has changed, the process releases access and backs out of the operation. If the frequency of concurrent access is low enough, then the probability is high that the process will be able to write the changes. With the right frequency of concurrent access, the two-phase commit exhibits significantly more throughput than using the brute-force method.

3. Protocols

Network protocols are generally modeled as processes passing messages to each other across some bi-directional media. The medium between two processes is modeled as storage accessible to both processes that holds the transmitted message for at least one snapshot. If the protocol is designed to work across faulty media, then in the formal model the storage is given the ability to generate spurious data, to lose data, and to corrupt data with nonzero probabilities. The procedure that the processes execute in order to send and receive messages is called the protocol. Generally, all processes execute the same protocol procedure, and the procedure tends to have a part that deals with sending and a part that deals with receiving. The two parts may even be two different invokable procedures. Usually, in an attempt to simplify the formal model while retaining its focus on the protocol, the formal model consists only of two processes, a sender executing only the sending part and the receiver executing only the receiving part of the protocol, and a uni-directional medium.

Main Properties to Prove

The main property that it is generally desired to prove about a protocol is that if a message m is sent from the sender, it eventually is received uncorrupted by the receiver. For some protocols, there may be additional properties, such as that the messages arrive uncorrupted in the order that they are sent.

The second of these properties is a safety property, because it says that in all snapshots, the sequence of messages sent by the sender is an initial subsequence of the sequence of messages received by the receiver. However, the main property is a liveness property, because it says that in all snapshots, if a message is sent by the sender in that snapshot, then there exists a later snapshot in which the same message has been received by the receiver. Hence, a popular way to specify and verify properties of protocols is with temporal logic [Hailpern82, Schwartz81, Lamport83a, Schwabe85, Wolper82, Nguyen84].

Temporal Logic Specifications

As an example, consider a general protocol specification given by Hailpern. In this specification, the medium is modeled as a monitor:

send(m):
pre: $\alpha=A$
post: $\alpha=A\langle m\rangle$
live: $\Diamond$(**after** *send*)

receive(**var** m):
pre: $\beta=B$
post: $\beta=B\langle m\rangle$
live: $\Diamond(ExistsM) \supset \Diamond$(**after** *receive*)

ExistsM:
pre: *true*
post: *true*
live: $\Diamond$(**after** *ExistsM*)

Monitor Invariant:
$ExistsM \wedge \Box(\neg$**after** *receive*$) \supset \Box(ExistsM)$

Note how a message cannot be received until after the message exists in the communication medium. The monitor invariant, relying on the assumption of only one receiver, says that if *ExistsM* becomes true, then it will not become false before a *receive* operation takes place, i.e., if an existing message is never received then it will always be existing.

The property that the medium can duplicate, lose, and re-order messages is captured by using history variables α and β which keep the list of messages sent and received respectively. Then it is specified that

$$m \in \alpha \supset m \in \beta$$

Thus no spurious and corrupted messages are created. In order that the protocol eventually deliver a message, it must be that the medium eventually transmits a message if it is sent often enough. This is guaranteed by putting two commitments on the medium, one for the send end and one for the receive end.

$$\alpha\ grows\text{-}without\text{-}bound \supset \Box\Diamond(ExistsM)$$

$$m\ repeated\text{-}without\text{-}bound\text{-}in\ \alpha \wedge \beta\ grows\text{-}without\text{-}bound \supset \Diamond(m \in \beta)$$

The first says that pumping enough messages into the medium guarantees that eventually one will exist in the medium. The second says that if a particular message is pumped into the medium often enough and the receiver receives often enough, then eventually the particular message will be received.

Hailpern's book describes two other protocols in detail and proves that each causes its medium to satisfy the service specification and the commitment of the medium described above.

Operational Specifications

There are a number of operational definitions of protocols. Among them are specifications written in AFFIRM [Sunshine82] and in a graphic specification language invented by Chen and Yeh [Chen83].

AFFIRM

The AFFIRM language has been used to specify a variety of protocols and the AFFIRM verification environment has been used to verify a number of properties of these specifications. A later section describes AFFIRM and its environment. For now, it suffices to say that AFFIRM allows specifications of NDISM in a way in which it is possible to quantify over snapshots. Therefore, it is possible in AFFIRM to specify liveness as well as safety properties. Sunshine, Thompson, Erickson, Gerhart, and Schwabe [Sunshine82] have written a paper in which they show how several protocols can be specified and verified to satisfy a number of useful properties.

They define a basic service protocol which captures what the protocol user sees, that is, as a reliable medium which delivers all messages, uncorrupted, and in the order sent. Typical of the properties of the service protocol that they specify are:

- Messages are delivered uncorrupted and in the order sent, a safety property.
- All messages that are sent are delivered, a liveness property.

These are specified using whatever needed quantification over snapshots. Then the AFFIRM natural deduction interactive verifier is used to prove that the service protocol satisfies these properties. Actually, failed attempts to prove these properties of the service protocol led to modifications to the specifications of both the service protocol and the properties. These modifications continue until the proofs had been successfully carried out.

Then they define a number of implementing protocols, such as the Alternating Bit (AB) protocol, whose job is to implement the service protocol given unreliable transmission media that lose messages, corrupt messages, generate noise, etc. They prove that the implementing protocol satisfies the properties of the service protocol by proving each of the service properties as theorems in the implementing protocol under the mapping from the implementing protocol data to the service protocol data. Interestingly, they did not take the approach of proving that the implementing protocol was a correct implementation of the service protocol. Proving only that the so-called implementation satisfies the desired properties directly takes less effort than proving that the so-called implementation is in fact an implementation and then to deduce by transitivity that it satisfies the properties.

Graphic Specification

Another operational approach is that of Chen and Yeh. They use a picture-based model that directly exhibits distribution of processes typical of many concurrent systems these days, especially protocols. One draws structure diagrams consisting of nested modules (boxes) connected by directed arcs emerging from and entering into modules and output and input sockets. Figure 6 shows a typical diagram consisting of a single module called *RT* with an input socket *I* and an output socket *O*. A module contains some computing agent. Messages of arbitrary data types flow along the arcs. The sockets represent sets of message arrival events at the boundaries of the module. The behavior of an innermost module is described by giving axioms relating the message arrival events at its own sockets. These axioms may express equality, arithmetic, Boolean, etc. relations between contents of messages as well as temporal and enabling relations on the events themselves. For example, when the function of the *RT* module is that an output message $o \in O$ that arrives at the output socket *O* is a copy of an input message $i \in I$ that arrived at the input socket *I*, then

1. the arrival, *i*, of the input message at the input socket *precedes* the arrival, *o*, of the output message at the output socket:

$$i \rightarrow o$$

2. *i* enables *o*:

$$i \Longrightarrow o$$

3. the contents of *i* and *o* are the same:

$$i.\text{cont} = o.\text{cont}$$

The language permits existential and universal quantification of events. Thus it is possible to express both safety and liveness properties. The "precedes" and "enables" relations on events are defined axiomatically. For example, "precedes" is defined as a partial ordering. This allows defining two events as concurrent if neither can be shown to precede the other. Once the model and its properties are specified, any theorem prover that can handle axiomatic definitions can be used to carry out proofs of property satisfaction by the model.

The paper gives a protocol service specification as a single module with one input and one output socket. The relations on the events on these two sockets describe a reliable transmission from the input to the output socket. The paper then gives a specification of the AB protocol as a more detailed nest of modules which contain an innermost module modeling an unreliable medium. The AB protocol is proved to implement the service protocol by proving the relations that specify the service protocol to be theorems in the AB protocol model. Also a number of the usual safety and liveness properties are proved of the AB protocol model.

4. Other Problems

Hayes has edited a collection of case studies of formal specifications [Hayes87]. Among the examples specified are a telephone network, the UNIX file system, several reservation systems, and several distributed systems.

V. Doing the Verifications

There is an excellent book by Howard Barringer [Barringer85] that surveys all the specification and verification methods except denotational and temporal. The papers by Lamport and Owicki [Lamport83a, Lamport83b, Lamport86, Owicki82] on the topic of temporal semantics show how to do temporal proofs. The survey article by Cheheyl, Gasser, Huff, and Millen [Cheheyl81] shows how the various operational systems are used. The detail required for an adequate coverage precludes their inclusion in this module.

VI. Specification Languages and Verification Environments

Many of the operational semantic languages have been implemented. That is, for these languages, there exists a specification language processor which given a specification and a list of properties to prove, generates conjectures, which if proved, imply the holding of the properties. Most of these systems also come with interactive or semi-automatic theorem provers that can be used to prove the conjectures generated from a specification. In fact, these tools sit in environments that are directed at specifying and implementing verifiably correct systems.

The specification and verification environments surveyed in this module are representative what is available:

- AFFIRM
- FDM
- Gypsy
- HDM
- P-NUT
- SARA
- PAISLey
- STATEMATE
- Process Algebras
- ASTRAL

Most of these environments have actually been used to carry out specifications and property verifications of real software. These include mainly kernels of operating systems that are supposed to enforce security policies and protocols.

For each language and environment the discussion includes descriptions of as much of the following as is relevant.

1. the language itself

2. how one specifies an NDISM in the language
3. how complete the specification can be
4. what kinds of properties can be proved of an NDISM
5. how one specifies, if at all, the redundant properties that are to be proved of an NDISM
6. the associated system development method
7. known limitations of the language or its environment
8. the tools of the environment
9. actual system developments to which the language and its environment have been applied

Topics 1, 8, and 9 get their own subsections and topics 2 through 7 are covered in a single subsection entitled "Specification and Verification".

Each of these is described by literature mentioned in its own description below. There are some surveys that cover more than one of these. The first four are surveyed very thoroughly by Cheheyl, Gasser, Huff, and Millen [Cheheyl81]. The Cohen, Harwood, and Jackson book [Cohen86] has brief descriptions of AFFIRM, Gypsy, HDM, and SARA. The Lindsey survey [Lindsay88] describes a number of automated verification environments, including AFFIRM and Gypsy.

1. AFFIRM

Language

AFFIRM [Thompson81] was designed by USC's Information Sciences Institute (ISI) as a system for specifying abstract data types (ADTs) [Liskov74] and verifying their properties. The AFFIRM language is basically a notation for writing algebraic specifications of abstract data types [Guttag78]. First, the signature of each operation of the ADT is given in the form of the list of the types of its parameters and the type of its return value. Then a set of axioms are given to relate the values of various applications of the operations.

The theorem prover uses the axioms defining an ADT as rewrite rules. Therefore, there are restrictions on the form of the axioms to make sure that their applications as rewrite rules cannot go into infinite loops. Among the restrictions is that the left side should consist of a single function application and that the right side should be have at least as many terms as the left side.

Specification and Verification

One defines an NDISM as an ADT whose data are the snapshots and whose operations create the initial snapshot and perform the individual transformations. That is, each operation except the create operation accepts a snapshot as its only parameter and returns as its result a next snapshot. The create operation takes no parameter and returns an initial snapshot. As there may be more than one operation applicable to a given snapshot, it is possible to model nondeterminism. The operations are specified algebraically, using equational axioms, whose left and right sides contain applications of operations, including the create operations.

The view of rules as rewrite rules is interesting when the ADT is used to define an NDISM. Specifically application of a rule can be viewed as a snapshot transformation, i.e., as taking one step of the computation. Thus a computation can be viewed as a sequence of rewrite rule applications. Concurrency is modeled when there is no unique sequence of rewritings, that is when which rule is applied next is nondeterministic.

The AFFIRM language permits universal and existential quantification over the values of the abstract data type. Therefore, it is possible to express both safety and liveness properties in AFFIRM specifications of NDISMs.

Tools

The AFFIRM environment contains a number of useful tools, including

- an algebraic specification analyzer,
- a library of useful data type specifications,
- a library of useful proofs,
- a theorem prover, and
- a friendly user interface.

The specifications analyzer attempts to check that the specification meets certain well-formedness properties such as suitability as rewrite rules and completeness. It can only attempt a completeness check, because in general completeness of an axiom set is undecidable. The theorem prover allows interactive natural deduction. The user enters the relevant ADT specifications and then gives it conjectures about these data types to prove.

Actual Use

AFFIRM has been used to specify and prove properties of a number of network protocols [Sunshine82]. The properties proved include both safety and liveness properties. AFFIRM was also used in the Delta Experiment [Gerhart79]. The Delta system is part of the Military Message System (SIGMA). It is a collection of functions, about 1000 lines of BLISS code, for reading and appending comments to messages in a secure manner. The collection was specified as operations of a message ADT. Certain security properties were proved. An abstract implementation was written and proved correct. This abstract program was generated by hand from the original BLISS language programs of the system.

2. FDM

The Formal Development Method (FDM) [Eggert89, Barton88, Eckmann89, Holtsberg89] was developed by Unisys (which absorbed System Development Corporation) for use in specifying systems and verifying that they satisfy desired properties The language of FDM is the Ina Jo language [Locasso80, Scheid86a, Scheid86b, Scheid89].

Language

The Ina Jo language allows a direct specification of NDISM by giving assertions relating before and after snapshots of individual selectable transformations. The language permits quantification over values of snapshot variables but not over snapshots themselves.

Specification and Verification

An Ina Jo specification consists of a list of level specifications. The first of these is called the Top Level Specification (TLS), and the last of these is called the implementation specification. Each level specification consists of four main parts

- data declarations
- initial conditions
- transform specifications
- correctness assertions
- mapping specification (except in the TLS)

The data declarations declare the types of values used by the variables and the variables appearing in the snapshot ("state" in FDM terminology), and the possibly parameterized snapshot interrogation functions.

The initial conditions describe by assertion all the possible legal initial snapshots.

The transform specifications describe for each transform the conditions under which it may be invoked and how the before and after snapshots are related; these relations can be functional, but they do not have to be.

The correctness assertions are redundant assertions of two kinds, criteria and constraints. Each criterion is a one-snapshot assertion that is supposed to be an invariant, and each constraint is a two-snapshot assertion that is supposed to be satisfied by all transforms.

All levels except the TLS have a mapping section which explains how to write expressions of the language of the preceding level, i.e., its parent level, in the language of its own level. This is to allow an assertion of the parent level to be rewritten in the language of the child level in order to see if it can be proved a theorem in the child level. In this manner, the redundant specifications, discussed below, of the parent level can be proved to hold in the child level.

The FDM suggests the following way of designing a secure system. The designer starts with the writing of the TLS. He or she decides on the data of its snapshot, on its initial conditions, and on its security policy as expressed by criteria and constraints involving the variables of the snapshot.

The designer then begins an iterative process by which transforms are added until the model has all the desired transforms and all of them preserve the criteria and satisfy the constraints. As a transform is added, it should be verified as preserving the criteria and satisfying the constraints. If not, then the transform must be changed until the verification succeeds. As the set of transforms is being decided upon, the designer may also choose to start supplying more implementation details. In this case, the designer specifies a new level in which the data variables exhibit more implementation details. The designer writes the entire level, the data, the initial conditions, and the transforms so as to implement the parent level. The way in which the child data implements the parent data is captured in the mapping section of the child level.

Because of the great difficulty in carrying out detailed proofs of implementation correctness, the designers of the FDM have stuck to more modest goals. Specifically, it is required only to prove that the child level satisfy the criteria and the constraints of the TLS that define the correctness of the model. Rather than prove the child level a correct implementation of the parent level through its mapping, each successive level starting with the child of the TLS is shown to satisfy the mapped criteria and constraints of the TLS brought down through its parent.

Consistent with this method of adding more transforms, the set of computations defined by an Ina Jo level specification is *not* necessarily that generated by applying arbitrary compositions of transforms to the snapshots satisfying the initial conditions. The set of computations is a superset of these and a subset of those generated by applying the arbitrary composition of the constraints, considered as transforms, to the snapshots satisfying the initial conditions. While it is never known if there will not be more transforms added later, it is known that all those added later will satisfy the constraints.

Because the redundant assertions are invariants and constraints, one can specify and prove safety properties and relations between successive snapshots. Because there is no quantification over snapshots, one cannot specify and prove liveness properties in the Ina Jo language.

Tools

The FDM is supported by an environment, the FDM platform, which includes a number of tools, the most important of which are the following:

- Ina Jo processor
- Interactive Theorem Prover (ITP)

The Ina Jo processor is essentially a compiler. It checks input Ina Jo specification for syntax and type errors, and if none are found, it generates a set of conjectures, one for each level specification. These conjectures imply that

- the initial conditions satisfy the possibly mapped criteria,
- the transforms preserve the possibly mapped criteria, and
- the transforms satisfy the possibly mapped constraints.

In other words, if the conjectures are verified, then the NDISM specified by the data declarations, the initial conditions, and the transform specifications have been shown to meet the correctness conditions as given by the redundant assertions.

The ITP [Stein80, Smith86] is used in order to attempt to prove these conjectures. The ITP first generates the logical negation of the conjecture to be proved; then the ITP helps the user find a contradiction thus proving the original conjecture a theorem.

Actual Use

Despite the original intention of being able to carry out the FDM over several levels down to code, no application has even been carried out over more than 2 levels and no application has been carried to code. In order to carry out the FDM all the way to code, another tool is needed for each programming language the might be used in conjunction with the FDM, namely a verification condition generator (VCG) [Scheid83a, Scheid83b] that generates verification conditions asserting that a given procedure body implements a given transform as specified in the implementation level specification. Unfortunately, to date, there is no complete VCG for any programming language.

The FDM has been applied to specify and prove the data security of a number of system programs. One of these was the AUTODIN II multi-level secure packet switching network [Cheheyl81]. The second of these was for KVM/370 a secure kernelized version of VM/370 [Gold79]. In both of these cases, the top level was proved to meet the specified security requirements. Neither was carried to any additional levels.

3. Gypsy

The Gypsy Verification Environment (GVE) was developed by a group of people at the University of Texas at Austin [Good78, Ambler78, Good84a, Good84b]. Whereas the emphasis of AFFIRM, FDM, and HDM are on verification that a specification and a design meet stated requirements, the emphasis of the GVE is on verification that an implementation is correct to stated requirements. The GVE has been used for code correctness proofs, which, in general have not been done in any actual application of AFFIRM, FDM, and HDM.

Language

The language of the GVE is Gypsy. Consistent with the emphasis of the GVE, Gypsy is a Pascal-like language that can be used both for specification and for implementation. There are restrictions from Pascal and extensions to Pascal in Gypsy. The restrictions are for the purpose of eliminating features that cause anomalies to the standard Hoare axiom system. For example, no function or procedure accesses any non-local variable; all accesses to a variable other than local variables must be via variable parameters. Functions have only by-value parameters and thus cannot have side effects. Procedures that do not return values can have side effects only through their parameters. On the extension side, there is a syntax for assertions that can be attached to arbitrary points in the program as well as at procedure and function headings as interface specifications.

Specification and Verification

In Gypsy one specifies NDISMs solely in an indirect manner by writing a program that implements it. In order to obtain concurrency, Gypsy has a number of features for setting up processes and for synchronization.

Tools

The GVE has a number of tools:

- Gypsy syntax directed editor
- Gypsy parser
- Gypsy interpreter
- Gypsy compiler
- Gypsy optimizer
- verification condition generator (VCG)
- theorem prover

Controlling all of these is the GVE executive. It is worth noting that the optimizer uses knowledge that can be gleaned from the assertions in making optimization decisions; for example, if the assertions imply that one arm of a conditional is impossible to ever follow, its code is not generated.

The VCG generates conjectures which imply that the code is partially correct with respect to the assertions given as annotations in the code. Because of

its basis on the Hoare axiom system, safety properties can be proved. This author believes that certain liveness properties such as halting are also proved, but not by conjectures generated by the VCG. These have to be submitted directly to the theorem prover

The GVE uses a Boyer-Moore theorem prover.

Actual Use

The GVE has been used for a secure network application [Good82]. That is the software that encrypts messages prior to sending them out over a non-secure network and that decrypts received messages only for the stated recipient. It was verified that only encrypted messages get out into the network, and that all received messages are equal to the corresponding original message. The GVE was also used to develop the ACCAT guard. The ACCAT guard is used to assist a human being in downgrading documents from a higher security level to a lower one. It is proved that no data are downgraded unless the human operator has seen it; thus it is presumed that the human is cleared to the higher security level and is competent to decide on the sensitivity of the document and to excise portions that should not be released to those who are cleared for the lower security level. For both of these projects the security criteria was stated as Gypsy assertions and the code was written in Gypsy.

More recently, the people responsible for the GVE have moved in other directions. One such direction is to use the Boyer-Moore logic directly to to specify both an abstraction and an implmentation as NDISMs and to prove the correctness of the implementation of the abstraction by showing that the latter simulates the former as suggested in Section III.1.d [Bevier87]. The Boyer-Moore prover has also been used to mechanize a proof system [Goldschlag90a, Goldschlag90b], defined in the Boyer-Moore logic, for UNITY [Chandy88].

4. HDM

The Hierarchical Development Method (HDM) was developed at SRI International for the purpose of specifying and verifying the security of operating system kernels [Robinson79, Silverberg79]. Operating systems are developed by hierarchical decomposition and the kernels are multi-level secure. The hierarchical decomposition of the system is carried out using traditional software engineering methods; for example, information hiding is used to decide how to decompose a system into modules, and each module is made an abstract machine providing some service, such as memory management, to the other abstract machines. Each such abstract machine is specified as an NDISM using SPECIAL.

Language

The specification language of the HDM is SPECIAL. SPECIAL allows direct specification of NDISMs. A SPECIAL specification of an NDISM consists of two main parts,

- the data part, and
- the algorithm part.

The data part declares types used by the variables and the variables of the snapshot of the NDISM. Each data type is specified algebraically with axioms.

The algorithm part consists of a collection of operation specifications. There are two kinds of operations, VFUNs and OFUNs. VFUNs are value functions; they return values computed from the current values of variables and have no side effects on these variables. OFUNs are procedures that change the values of variables and do not return values. Their notation is not unlike that of Parnas's [Parnas72b], and indeed the HDM designers suggest using the same software design methods touted by Parnas in the same and related papers [Parnas72a].

Specification and Verification

In the HDM terminology, the specification of an NDISM is a Top Level System (TLS). The TLS specifies the visible behavior of the system, which the users of the abstract machine can rely upon. Below the TLS, is a list of NDISMs each of which is a refinement of the one above it. A refinement NDISM provides more implementation details than the one above it, but provides *no* new operations; to do so would violate the TLS's specifying the visible behavior of the system. Below the lowest level comes the implementation of the specified TLS as a module or a collection of procedures and data structures in some implemented programming language.

Tools

There are no redundant properties mentioned in a SPECIAL specification. Instead the properties to be proved are provided by the prover at proof time to the verifier. Given the basic purpose of the HDM to be able to prove systems multi-level secure (MLS), there is an MLS Formula Generator which takes a TLS and generates conjectures which collectively imply the multi-level security of the TLS. The MLS Formula Generator is based on Feiertag's [Feiertag80] model of multi-level security.

These conjectures can be submitted along with the TLS to the Boyer-Moore theorem prover that is available. The user of the theorem prover may also interactively provide other conjectures to prove, thus allowing him or her to prove properties other than multi-level security.

While the TLS is to be demonstrated MLS, the refinement levels are to be proved correct implementations of their immediate parents. This proof is based on the standard mapping from the implementing level's data to the implemented level's data described in Section II.1.d. In addition the code implementing a TLS is to be proved correct by proving each procedure a correct implementation of the corresponding operation in the lowest level refinement of the TLS. For each implementing programming language there is supposed to be a verification condition generator (VCG) that generates conjectures implying this correctness relation. At present there are VCGs for Pascal, FORTRAN, and JOVIAL. In order not to have to develop a VCG for every programming language that might be used, SRI has developed a Common Internal Form (CIF) which can express implementations and whose programs can be translated straightforwardly to any implementation language. They have written a VCG for CIF as the implementing language. Currently translation from CIF notation to an implemented programming language is done by hand.

Actual Use

The HDM has been used to specify, design, and verify MLS secure two operating systems, Kernelized Secure Operating System (KSOS) [McCauley79] and Provably Secure Operating System (PSOS) [Neumann77, Feiertag79].

5. P-NUT

P-NUT is a suite of tools developed by the Distributed Systems Project at University of California at Irvine [Razouk85a, Razouk85b, Morgan87]. The purpose of these tools is to allow design of concurrent systems in a manner that allows detection of timing, synchronization, and resource contention errors. The approach used is to specify these systems with analyzable models and to have tools which do the analysis. P-NUT is built around giving Petri net models of concurrent systems [Peterson81].

Language

A Petri net is a bipartite directed graph whose nodes are *transitions* and *places*. Figure 7 shows a Petri net describing a service protocol. The places $P1$ and $P2$ send messages to each other via the transitions $T1$ and $T2$. The directed arcs indicate for each transition which places are input places and which places are output places, namely those places which are at the tail and heads of the arcs respectively.

A snapshot in a Petri net computation consists of an assignment of zero or more tokens to each place. A transition is fireable if there is at least one token on each of its input places. If a transition fires, then one token is removed from each input place and one token is deposited at each of its output places to create the token assignment of the next snapshot. At each snapshot any fireable transition may be selected for firing. Thus concurrency is modeled by having more than one transition fireable.

A Petri net may be timed or untimed. Normally a Petri net is untimed. When it is timed, a delay is associated with each transition indicating how long it takes for the transition to fire. A Petri net may also be interpreted or uninterpreted. Normally it is uninterpreted, but one may associate an actual function with a transition. In this case, each token represents one parameter. Thus these functions are not applied unless all input parameters have been supplied.

Specification and Verification

A system is specified by giving a Petri net for it. This Petri net can be totally uninterpreted; however, then not much beyond its flow control is specified. More information can be given in the form of timing estimates for the firing of places and interpretations, i.e., procedures, that explain what happens when a place fires.

Tools

The P-NUT environment contains a variety of tools, including

- an interactive graphic Petri net editor,
- a translator,
- an untimed reachability graph builder,
- a timed reachability graph builder,
- a reachability graph analyzer,
- a reachability graph pretty printer, and
- a simulator.

The translator converts a Petri net input in linear form into an intermediate form which is used by all the other tools to stand for the net.

The reachability graph builders build graphs whose nodes are all the snapshots achievable in any computation of a candidate Petri net. The builders try to keep these graphs finite by recognizing when a newly generated state is the same as one it has seen before.

These reachability graphs are analyzed for timing, synchronization, resource contention, and deadlock problems by the reachability graph analyzer (RGA). The RGA is the most important tool of P-NUT. With it, the user can specify propositions and predicates about places, transitions, and arcs in the Petri net and about reachable states in the reachability graph. All of these entities can also be bound by both the existential and universal quantifier. Therefore both safety and liveness properties can be specified. The RGA proves these by exhaustive examination of the reachability graph. There are also a num-

ber of generic properties that the RGA knows how to demonstrate:

- boundedness, that there is some constant maximum number of tokens at each place
- safeness, that there is a maximum of one token at each place
- conservation, that the total number of tokens in any snapshot is less than or equal to some constant
- liveness, that a particular transition is potentially fireable in all reachable snapshots
- deadlock-freeness, that it is not the case that no transition is alive

The simulator takes a graph in intermediate form and an initial assignment of tokens and simulates computations by firing fireable nodes and redistributing tokens according to the firings. If the net is timed the simulator will keep track of simulated time. If the net is interpreted, the simulator will invoke the procedure for fired nodes and keep track of the values of the tokens. There are useful facilities for controlling the progress of the simulation. The user may choose which node to fire, or he or she may let the simulator select one at random.

The interactive graphic Petri net editor allows inputting nets by drawing them with the aid of a mouse. This is more convenient than the linear form of the graph consisting of ordered pairs defining the arcs.

Actual Use

The tools of P-NUT have been used to specify protocols and to prove both safety and liveness properties about them. The paper by Morgan and Razouk [Morgan87] shows a specification of the Alternating Bit (AB) protocol and proof of its properties. There is a service specification as a fairly simple Petri net and then a very detailed Petri net describing the AB protocol as an implementation of the service specification. With the help of the RGA, the AB protocol Petri net is proved to satisfy the liveness property of implementing the service specification. The AB protocol Petri net is also shown to be safe, so that the readers are mutually exclusive.

6. SARA

The SARA design environment was developed at UCLA to support requirement-driven top-down design of possibly concurrent digital systems [Razouk77, Vernon80, Razouk80, Estrin 86]. The SARA method dictates top-down design from requirements until the level at which it is possible to compose the system from off-the-shelf components.

Language

At the topmost level or at each refinement of a module, one gives multiple models of the system or module at hand, each from a slightly different point of view. The models are structural and behavioral, and there is a mapping between behavior model elements and the supporting structure model elements. A structure model (SM) consists a number of nested modules (boxes) connected by undirected arcs that meet the modules only at sockets. Figure 8 shows an SM on the lefthand side with two modules, *M1* and *M2*, and an arc connecting them at sockets. Observe that the modules are mapped via dashed lines to elements in the rest of the diagram. The behavior model, in the form of a graph model of behavior (GMB) consists of three submodels,

- a control flow graph (CFG),
- a data flow graph (DFG), and
- an interpretation.

The two flow graphs together form an uninterpreted model. Figure 8 shows a GMB on the righthand side.

The CFG consists of process nodes connected by directed arcs. An arc represents precedence of activation of the processes at the head and tail; that is, the process at the tail must finish *before* the process at the head can begin. Each CFG node may have more than one in-pointing arc headed at it and more than one out-pointing arc tailed at it. For a given node, the in-pointing arcs are either in an "and" or an "or" relation, and likewise for the out-pointing arcs. These relations constitute the input and output logics for the process node, respectively. The CFG in the GMB of Figure 8 has two process nodes, *P1* and *P2*, connected by two arcs flowing in opposite directions.

A snapshot in the computation of a CFG is an assignment of zero or more tokens to each arc. In any snapshot a process node is enabled if the tokens on its in-pointing arcs satisfy the logic; that is, if the logic on the in-pointing arcs is "or", then at least one such arc has at least one token, and if the logic is "and", then all such arcs have at least one token. In each snapshot, any enabled process node is selected for initiation, and execution. The next snapshot is obtained by removing tokens from the initiated process's in-pointing arcs, according to the input logic, and depositing tokens on the process's out-pointing arcs, according to the output logic. That is, if the input logic is "or", then exactly one token is removed from exactly one of the in-pointing arcs, and if the input logic is "and", then exactly one token is removed from each of the in-pointing arcs. Similarly, if the output logic is "or", then exactly one token is deposited on exactly one of the out-pointing arcs, and if the output logic is "and", then exactly one token is deposited on each of the out-pointing arcs.

The DFG is a bipartite composed of process and dataset nodes. An arc with the head at a process represents the dataset at the tail being an input to the process, and an arc with the tail at a process represents the dataset at the head being an output of the process. Each DFG process is associated with some process in the CFG so that a DFG process is initiated to do its data transformation from input to output when the corresponding CFG is initiated. The DFG of the GMB of Figure 8 has two process nodes, *P*1 and *P*2, and two dataset nodes, *D*1 and *D*2. All the arcs flow from one kind of node to the other. Observe that an association has been set up mapping the DFG processes *P*1 and *P*2 to the CFG processes *P*1 and *P*2 respectively.

Associated with each process node in the CFG may be an interpretation, a procedure in some implemented programming language, LISP in the current version of SARA, describing the behavior of the process. The input and output of this procedure must be consistent with the connectivity of the process's associate in the DFG.

Specification and Verification

The design and implementation of a system proceeds from requirements downward to implementation using components, including programming language statements, off the shelf. The requirements are stated as an enumerated list of natural language sentences, the numbering giving a means to map implementation details to the requirements that they help implement. The topmost level design is a complete complement of designs in each of SARA's languages. The tools described below give means to analyze, simulate, and test the design. As the designer moves from level to level, he or she takes a yet unimplemented SM module and either implements it easily with off-the-shelf components or refines it into a complete complement of models giving more details. This process continues until the system is completely tested and implemented.

Tools

The SARA environment is a graphics based environment in which one can draw the models and establish the mappings and associations between elements of the models visually. Figure 8 shows these mappings as dashed lines. In this environment, there are a variety of tools that can be used to validate the models, in particular the GMB models,

- GMB Analyzer for analyzing the CFG in order to test for proper termination and thus for deadlock,
- Performance Analyzer for a stochastic variant of the CFG, and
- GMB Simulator for allowing the user to exercise the model with visual feedback.

The GMB Analyzer generates the graph of all reachable states and then asks various questions that can be answered by examining the structure of this graph.

The Performance Analyzer uses supplied timing information to build a closed form queueing theory model of the CFG.

The GMB Simulator is in the form of a token machine. It implements the token-distribution semantics of the GMB model, allowing the user to choose among enabled nodes, and it invokes associated interpretation procedures as process nodes are initiated. The token machine shows the simulation on the screen by highlighting initiated nodes and showing tokens moving from arc to arc.

Actual Use

The models of SARA have been used to describe protocols, and the tools of the SARA environment have been used to verify certain useful properties of these protocols, such as proper termination [Razouk79, Ruggiero79]. The SARA tools do not permit verification of correctness, since the requirements language is only natural language.

7. PAISLey

PAISLey [Zave72, Zave87a, Zave87b, Zave87c, Zave91] is an executable specification language for specifying concurrent digital systems. That is, besides formally specifying a concurrent system, a PAISLey definition can be given to an interpreter that will execute the specification to produce one or more of the specified computations.

It is claimed that PAISLey is different from an ordinary programming language because a specification in PAISLey, when it can be given, is implementation independent. That is, it can describe all required properties and behavior of a digital system only to the level of detail desired, leaving all other aspects, including how the properties and behavior are implemented, unspecified. For example, instead of describing a computation over all the elements of a list as a loop that visits the elements in some order, the result of this computation is specified using universal quantification over the elements of the list

PAISLey tries to get the best of two computational models, asynchronously interacting concurrent processes and functional programming. The cycle of each process is specified functionally. One can specify timing constraints on any function, and thus also on any step of processes. These timing constraints may either be a formula giving the estimated time of a step or formulae giving upper and lower bounds on the estimated time of a step. The interpreter keeps lower and upper bound total time estimates for any computation path it follows.

According to Zave and others who have used PAISLey, PAISLey is intended to and is well-suited to model highly parallel, real-time, possibly distributed systems, such as process-control and communication systems.

Language and Specification

A PAISLey system is a fixed set of processes, each running independently and asynchronously with respect to the others of the set. By "fixed number of processes" is meant that the number of processes is determinable at specification processing time. In order not to have to write many nearly identical copies of a definition that is describing the behavior of more than one process, a process specification can be replicated over an indexed set of processes, but the replication is a kind of macro obeyed by the specification processor. The purpose of boundedness in the number of processes is to allow proof that synchronization attempts will be successful within a bounded amount of time, by allowing establishing bounds on the mount of time required to perform any observable function.

Formally a system is specified as a tuple of process cycle functions, each applied to its initial state. Thus to specify a two-process system, consisting of a clock whose state is the current time and a watcher whose state is the last observed time, one would write

```
TIME=INTEGER; clock-cycle:
    TIME --> TIME;
    watcher-cycle: TIME
       --> TIME; (clock-cycle[0],
                watcher-cycle[0]);
```

A process is specified with the aid of a cycle function that maps the process's state to its next state. Each application of the cycle function is called a *process step* or a *process cycle*. Because the argument to a process's cycle-function is only the process's own state, each process appears to have a non-forked, deterministic computation tree. However, processes do communicate by exchanging values through shared synchronizing channels. That is, a process may have to wait unspecifiable numbers of other process's steps before it can continue its own step with an unpredictable value. The unpredictability in the number of steps and in the value returned from the exchange causes nondeterminism.

In PAISLey, communication between two processes occurs with the help of *exchange functions*. Each exchange function name specifies both the type of exchange and the channel over which the exchange occurs. For example, `x-time` specifies an x type exchange over the time channel. When two processes have called an exchange over the same channel at the same time (defined more precisely below) then the function calls are said to match, and each process returns as the value of the function call the argument of the other's call. That is, if process Π_1 calls `x-time[1]`, and process Π_2 calls `x-time[2]` at times that allow the calls to match, then Π_1 returns 2 from its call of `x-time`, and Π_2 returns 1 from its call of `x-time`. To each process, an exchange function call appears as an ordinary value-returning function call; however, during the execution of the exchange function call, depending on the type of the function, a process may have to wait asleep, until a match occurs between exchange function calls to the same channel.

The three types of exchange functions are x xr and xm. They differ by whether the calling process waits or not and with what other kind of exchange calls they can match. Each side of a communication may be of a different type.

- In an x-type function, the caller waits if there is no other process that has called an exchange function over the same channel. If and when a match occurs, the call returns the argument of the matching call.
- In an xr-type function, the caller does not wait if there is no other process that has called an exchange function over the same channel; in this case, the call returns the value of its own argument. If there is another exchange call to the same channel waiting or issued at the same cycle, the a match occurs, and the call returns the argument of the matching call.
- An xm-type function call behaves as an x-type function call, except that it cannot match another xm-type call on the same channel.

Thus x's can match x's, xr's, and xm's, xr's can match x's and xm's, but xm's can match only x's and xr's.

Any time an exchange function is evaluated, if more than one match is possible, then some legal match is chosen nondeterministically.

Consider the following PAISLey definition, of a clock ticking every second and a watcher occasionally looking at the current time of the clock. The current time is the state of the clock and the last time read from the clock is the state of the watcher. In PAISLey, comments are surrounded by quotes.

The cycle of the clock process can be described in words as: compute a new time value as the current time plus one and offer the current time value to anyone who will read it. The new time value, i.e., the first component of the tuple, is taken as the next state of the clock. The offering of the current time is accomplished as a call to an xr, non-waiting exchange function on the time channel that is read by the watcher. Since the clock does not wait, it can guarantee to finish its cycle in precisely one second.

The cycle of the watcher can be described in words as: wait until there is a current time being offered in the time channel; when the match occurs take the time argument passed into the matching call and return it as the read time. Note that when the watcher gets back a time, and that time is installed as the next state of the watcher that time will be one second behind the time installed as the next state of the clock.

"BEGIN DEFINITION"

```
TIME=INTEGER;
```

"INITIAL STATE OF PROCESSES" "It also says that the system has two processes, the clock and the watcher, each with its own cycle."

```
(clock-cycle[0], watcher-cycle[0]);
```

"DEFINITIONS FOR THE CLOCK PROCESS"

"The function clock-cycle takes TIME to the next TIME. Thus, the state of the clock is its current time, and its initial state 0 is time 0."

```
clock-cycle: TIME --> TIME;
```

"The lower bound and upper bound of the time for a cycle are both 1.0 second. Thus, a cycle is precisely 1.0 second long!"

```
clock-cycle: ! lb 1.0 s, ub 1.0 s;
```

"The next time is the first component of a two-tuple consisting of

> the current time bumped by 1
> and an offering of the current time to
> the watcher

i.e., the offering is done only for effect and its results are ignored in the clock's progression to its next state."

```
clock-cycle[time] =
        proj[(1,
                (sum[(time,1)] ,
                offer-time[time])
            )];
```

"offer-time either returns with the time it offered, indicating that this offered time was not read or it returns null, the unique value of type FILLER, indicating that the offered time was read."

```
offer-time: TIME --> TIME | FILLER;
```

"offer-time is defined as a more mnemonic name for a exchange function of type xr on the channel called time, i.e., for xr-time."

```
offer-time[time] = xr-time[time]
```

"It is important that the exchange function for offer-time not get hung up waiting if there is no other process who has asked to read the time from the time channel; if the clock were to get hung up, the clock would not be able to guarantee that it will finish an execution of clock-cycle in the 1.0 second time limit set above; thus its exchange function must be of type xr."

"xr-time will return its argument if no other process has asked to read the time channel and null otherwise, as is required for offer-time."

"DEFINITIONS FOR THE WATCHER PROCESS"

"watcher-cycle appears to take a TIME to some TIME, but in fact, it ignores the input TIME and occasionally returns a TIME read from the time channel."

```
watcher-cycle: TIME --> TIME;
watcher-cycle[time] =
        read-time[null];

read-time: FILLER -> TIME;
```

"read-time is a mnemonic for a waiting exchange function, i.e., of type x, on the time channel; x-time will wait until another process has attempted an exchange on the same channel and will return the argument of that call. Since the only other process, the clock-cycle, will be calling offer-time every second, the watcher's x-time will wait no more than one second until it returns, if and when the watcher does a call to x-time."

```
read-time[null] = x-time[null];
```

"END DEFINITION"

Now the question is what is the computational model of the formal model of PAISLey. That is, how is concurrency modeled? The model is unusual in that there are definitely sequences of states. However, the state transitions are not the unit of interleaving. Rather, a state transition is defined as an application of a function, which in turn may be defined in terms of other, possibly concurrent function applications. The initiation and termination of functions applications that are the units of interleaving.

A system in PAISLey is a set of processes, each running independently of and asynchronously with each other. A process specification describes a sequence of states. Each process has an initial state and each subsequent state is obtained from the previous by application of a *mapping* or function on states to states. Each application of a process's state transition mapping is called a *step* or *cycle* of the process. Figure 9(a) shows a system with three processes. In the absence of synchronization, each process's cycles proceed independently and concurrently of each other. Nothing can be said in the formal model about the speeds of these cycles. In

the absence of synchronization, the cycles do not occur in any kind of lock-step with each other. Thus, the figure shows the cycles taking varying amounts of time and states of processes not lined up with each other in time.

An *event* is either an initiation or a termination of a mapping application. Thus there is an event at least at the beginning and end of each step of any process. There may be more events, because the definition of a mapping may be in terms of other mappings or of tuples of mappings. The initiation and termination of each of these submappings is an event. Moreover, the individual mappings of a tuple of mappings are evaluated concurrently. Therefore, within each cycle of a process, there may be any number of mapping evaluations going on concurrently, with their starting and termination events occurring in any which order. Figure 9(b) shows the detail of two cycles of one process. Each cycle has an encompassing mapping application. Each of these contains nested mapping applications, some disjoint and some overlapping with each other. The events are marked with little filled boxes.

The events of all processes of a system are interleaved. Thus the smallest observable grain of computing are the initiation and the termination of a function evaluation. Within each process's cycle there may be any number of possibly concurrent function applications going on. It is their startings and endings that are interleaved. If two events are supposed to occur at precisely the same time, whatever that means, one of them will be chosen arbitrarily to execute first. Thus, in Figure 9(b), the two events on the same vertical, time axis near the middle of the second cycle are arbitrarily interleaved.

The author of this module asked Zave what is the model. She replied by electronic mail that "It is definitely operational, as there is an interpreter which nondeterministically selects the next event in the computation. Petri nets and dataflow diagrams both capture some aspects of the execution model."

Tools

As PAISLey is an executable specification language, there is an environment providing a specification checker and a specification simulator. The specification checker does all sorts of compile-time checks including that all functions are called with arguments of the correct type. The simulator tool is quite spiffy with a variety instructions to allow user to select more or less output, frequency of sampling, etc. Also the output is always pure ascii to allow other tools to be used to analyze the output in the traditional UNIX pipe paradigm.

Performance information can be given for functions. One can specify a distribution of execution times, lower and upper bounds on the execution time, or both. The simulator will keep track of simulated time elapsed, using a random number generator to select some time in the range if a range is given.

The PAISLey processor and other tools are available from the UNIX tool chest and is available free to educational institutions.

Actual Use

PAISLey has been used in projects at AT&T as an executable specification language [Zave87c, Zave91]. These include the specification of the interface to a database in the Submarine Lightguide Project, the specification of the user interface to the PAISLey environment, and, in the FEARS (Finite Element Adaptive Research Solver) project, the specification of a distributed implementation of an application that had never been distributed before. In all of these cases, it was found that "a little formalization goes a long way" [Zave91] to help understand difficult problems. This understanding comes from the mere exercise of trying to formalize the system under design *even* if the formalization is left incomplete.

Other Similar Systems

The specification language Gist [Feather87], developed by Feather and others at ISI, is in a family with PAISLey, although its formal definition is probably more carefully laid out. In Gist, the meaning of a concurrent program is a set of *histories*, each being a sequence of *transitions*. A transition is a set of *deltas*, and each delta is a function on a primitive state variable to its next value. By having a function for each variable of the state, concurrency can be represented to the granularity of the individual variable update. Gist is used to write specifications for components of distributed composite systems, from which correct implementations of the components and of the system itself can be formally derived.

Another operational approach based on sequences of events is that of TSL-1 (Task Sequencing Language-1) [Luckham86] TSL-1 is language for specifying sequences of tasking events that occur during the computation of a distributed Ada program. The events of interest are task initiations and terminations and rendezvous initiation and termination.

TSL-1 statements are annotations added to Ada program that appear as comments to the Ada compiler, but when processed by the TSL compiler, cause generation of calls to a run-time monitor which checks that the actual computation is consistent with the annotations.

TSL-1 statements allow specification of the events that will occur during a program's computation; it

also allows specification of when two of these events are *connected*, i.e, that one must follow the other. For example, the acceptance of an entry call is connected to and must always follow the corresponding entry call. The ordering of all other pairs of events must be left unspecified because in a distributed system, there is no clear notion of what happens first. Two events may appear in opposite orders to different observers at two different sites in the system. The specified events and connections imply a partial ordering of events. The checking done by the run-time monitor is that the actual events in the monitored computation are consistent with this partial ordering.

Because TSL-1 is formally defined, it should also be possible to formally verify the consistency of the specification with the program, and not have to rely on run-time monitoring.

8. STATEMATE

STATEMATE [Harel88a, Harel88b] is a graphics-based environment for describing reactive concurrent systems. The mode of use of it is similar to that of SARA, described above. One is expected to use the STATEMATE languages to carry out the design of a system starting from requirements. In STATEMATE, the system under design (SUD) is described with three different views, structural, functional, and behavioral. As with SARA, the idea is to obtain redundancy, as each aspect is described from three different points of view and these descriptions are reconciled as part of the consistency checks performed by the STATEMATE tools.

The structural view decomposes the SUD into its physical components, modules together with channels for the flow of information. The functional view describes what each component can do, i.e., the function it can compute or the data it can carry. The behavioral view describes the order in which the components are activated in order to carry out their function.

Language

For each of these three view, STATEMATE provides a graphical language together with an on-line graphics-based editor that checks the validity of a diagram as it is being built element by element. The resulting diagrams, called module-charts, activity-charts, and statecharts, respectively are all based on a number of new graphical conventions invented by the authors of STATEMATE to get around the size problems that other graphic notations have run into and which prevent them from being used on large, real systems.

As mentioned, STATEMATE was designed to allow specification of large and complex reactive systems. A reactive system, as opposed to a functional transformational system, such as a batch formatter, is event driven and must be continually available to react to internal and external stimuli. Examples include avionics systems, communication networks, computer operating systems, process controllers, telephones, VLSI circuits, windowing systems, WYSIWYG formatters, etc. The problem in designing such systems is to describe their behaviors clearly and realistically. The authors of STATEMATE have paid close attention to the problems of visual formalisms in designing their notation for the three views. They were quite creative in their notation for the behavioral descriptions.

To give more detail about the three views:

A structure chart for the SUD consists of

- possibly nested modules, each being a rectangle,
- information flows from modules to modules, each being a possibly multitailed and a multiheaded arrow,

The SUD is itself a module, and modules of the environment external to it are drawn with dashed lines. Inside the SUD module are both processing modules and data modules, with the latter being drawn with dashed lines. A module may encapsulate internal modules which are shown in another diagram in which the encapsulating module is regard as the SUD of that diagram. In this manner, the structure is hierarchically decomposed into modules that have no internal substructure. For an example, see Figure 10.

An activity chart is similar to a structure chart, except that the rectangles denote the activities or functions carried out by the system, and the arrows denote flows. A solid arrow denotes data flow and a dashed arrow denotes control flow. For an example, see Figure 11.

The behavior charts are the most innovative. They may be thought of as extended state transition diagrams (extended finite-state machine) for which the traditional limitations of these diagrams have been avoided. State transition diagrams are inappropriate for describing behavior of complex reactive systems because they are flat and unstructured and inherently sequential. Moreover, they tend to suffer an explosive, exponential growth in the number of states as the SUD is extended slightly.

These problems are avoided by the ability to hierarchically decompose states into AND- and OR-combined states plus a broadcast mechanism. Once states are hierarchically decomposed, it is necessary to be able to have transitions enter and leave at any level of the decomposition. Figure 12(b) shows the diagram resulting from OR-decomposition of Figure 12(a). Either diagram shows that from state V the computation goes either by transition e to state S or

by transition *h* to state *T* and then from either back to *V* by transition *f*. Figure 12(b) has clustered states *S* and *T* into a new state *U* such that to be in *U* means being either in *S* or *T*. If it turns out that *T* is specified as the default state of *U*, then the diagram can be changed to that of Figure 12(c).

Figure 13 shows AND-decomposition. In Figure 13(a), there is a cluster of ordered-pair states. If these are clustered into a new state *U* consisting of a Cartesian product of smaller state transition diagrams, the diagram can be simplified considerably into that shown in Figure 13(b). Quite likely the meaning of the individual sub state transition diagrams is clearer than that of the larger diagram with ordered-pair states. Recall your own experience constructing the AND- and OR-combined finite state machines when you learned that the union and the disjunction of finite state languages were also finite state. Figure 14 shows the use of broadcasting. In this machine, each state is a three-tuple. When the computation is in state *(V,W,R)* and event *m* occurs, then the next state is *(X,Y,P)* because the transition under *m* from *R* to *P* broadcasts *e*, causing transitions from *V* to *X* and from *W* to *Y*.

Concurrency is achieved through the AND decomposition using Cartesian product states. Each sub transition diagram can be considered the specification of an independent process running concurrently with the process that is specified by the other sub transition diagrams. Thus in Figure 13(b), one process is running the diagram in *S* and the other is running the diagram in *T*. In figure 14, there are three processes, *S*, *T*, and *Q*. Synchronization is said to occur at like named transitions in the sub transition diagrams. Thus in Figure 13(b), the first process that gets to a transition *e* will have to wait until the other gets there also in order that both can proceed.

The ability to decompose states and to broadcast are powerful tools for reducing the size of transition diagrams into manageable chunks, each part of which can be understood independently for its contribution to the whole. But what is the meaning of a computation when states have substructure. If a state *S* consists of a subdiagram, then activation of *S* means activation of the subdiagram. Then suppose *S*, at the level it appears atomic, is concurrent with state *T*, then what is the relationship between the components of *S* with *T*? Moreover, in an interleaving model, what is considered atomic is critical, for that determines the grain of interleaving. If in a nondeterministic selection *S* is selected to run before *T*, then should the submachine of *S* finish entirely before initiating *T* or should the components of *S* be interleaved with *T*?

The module author raised this question with the authors of STATEMATE and received an answer from Pnueli by electronic mail. The basic idea is to consider the atomic transitions, i.e., those that go from atomic states, which have no components. "At each step in the execution of a statechart, there is a set of events and conditions that come either from the environment, or have been generated by the statechart in the immediately *previous* step. These events and conditions *enable* a certain set of transitions, which are edges in the statechart graph that depart from a state that is currently active, and whose labels are satisfied by the events and conditions that are currently true. The next step consists of a maximal conflict-free set of enabled transitions, and all of them are jointly taken in this step. In case of nondeterminism, several such maximal conflict-free sets may be available." One of these is nondeterministically selected and all of its transitions are take in this step. "In interactive simulation, the user is asked to choose one. In non-interactive (batch-)simulation, the system chooses one at random. The taken transitions usually generate events and modify conditions, and these will be available in the construction of the next step."

There is true concurrency in the formal model, as for PAISLey. However, since the set of transitions selected to fire in parallel are mutually nonconflicting, the result can be no different that having gone through the same transitions in any order. Thus, the model is equivalent to a fully atomic-step *interleaving model*. The obvious question is "Why is there this unnecessary concurrency in the formal model?" Perhaps, it is to justify the interpreter tool using the same model, i.e., of doing many steps concurrently. Certainly time estimates will be more accurate if the concurrency in the model is an accurate reflection of the concurrency that can take place in real life.

Actual Use

STATEMATE is in use by several companies, including Israel Aircraft Industries and SEI itself to help specify and design reactive systems. Leveson and others report on the use of STATEMATE to build a system requirements specification for a real aircraft collision avoidance system, a system involving real-time concurrency [Leveson91].

Tools

Among the STATEMATE tools are the following:

- a simulation package for executing the specification of the SUD to allow observation of its behavior
- report generators that, among other things, can generate DoD required standard documentation
- testing package for doing a number of dynamic tests, such as reachability analysis and detection of deadlock and nondeterminateness

- an Ada code generator that produces an Ada prototype of the SUD according to simple rules about how to implement the components of a specification

9. Process Algebras

CCS (Calculus of Concurrent Systems) and CSP (Communicating Sequential Processes) are formalisms for specifying concurrent systems in terms of sets of possible traces of observable events given rise to by the system. Thus the formalisms are operational. Furthermore, each specifies a system in terms of its component processes that are independent except for explicit communication between them. Each component is specified in terms of constraints on its possible traces of observable events. The system as a whole is understood as a composition of these traces according to various trace composition operators.

In both, each process is specified in a grammar-like notation in which the nonterminals or variables are regarded as process states and the terminals or constants are regarded as names of actions, events, or transitions. A multiprocess system is specified by composing process definitions using a variety of operators that model choice between processes and concurrent execution of processes. These operators form a process algebra

CCS was developed as a formalism for describing multiprocess systems and for exploring the various notions of equivalence of processes [Milner80,Milner89]. The original CSP was in fact a programming language [Hoare78] that later proved to be the inspiration for the Occam language. A proof system for that version of CSP can be found in a paper by Apt, Francez, and de Roever [Apt80]. The CSP formalism (as opposed to programming language), developed to explore specification of processes, is described in a book of the same title [Hoare85]. These formalisms have been the inspiration for at least one standardized concurrent system specification language, LOTOS [ISO89] that is being used to specify Open Systems Interconnection (OSI). There is a more recent extension of CCS called SCCS (Synchronous CCS) for specifying sets of processes that operate in lock-step synchronized concurrency [Cohen86].

Language and Specification

The languages of CCS and CSP are built on the standard notation for mathematics, using theusual logical, set, and function operators.

In both (although strictly speaking, CCS does not really define the trace), a trace is a finite sequence of symbols representing atomic, indivisible observable events in which a process or processes have participated in up to some instant of time. The event symbols are uninterpreted in the CCS and CSP formalisms, but can be chosen to be mnemonic of a meaning to the human reader. For example, the event *start_timer* can be understood as the event of the timer starting up, even though to CCS and CSP, it is no more than just another atomic event.

Thus, in addition to the logic, set, and function operators are a collection of operators for working with finite sequences such as concatenation, head, tail, i^{th} element, etc.

The concept of *observable* event is critical and provides a powerful abstraction tool. It is up to the definer of a system to specify which of the possible indivisible actions that occur during the execution of the system are considered observable. Events not critical to the abstract description of the system can be ignored. Among the operators of CCS and CSP is restriction of a trace to symbols in a sub-alphabet of symbols.

The specification of a sequential program is given as a description of the set of all possible traces of observable events of the program. The specification of a concurrent program is given as the composition of specifications of its constituent sequential processes. This composition effects an interleaving of the events of the individual processes. For this interleaving to be meaningful and to capture all possible computation histories, it is essential that the events be atomic. Communication between two processes is accomplished by having two processes execute two separate halves of what is considered one event. One half is the write and the other half is the read of a datum on a channel. Both processes have the same event symbol in their traces. The process that arrives at its half of the event first must wait until the other arrives at the other half; only then does the event happen, i.e., the reader reads what is written.

Fundamental Semantics

In both, one views a collection of process definitions as specifying what is called a *synchronization tree*. A synchronization tree, shown in Figure 15, is not unlike the tree of possible computations of an NDISM illustrated in Figure 1. There is more information embodied in the synchronization tree, but it is possible to derive from it the tree of possible computations of the NDISM implied by the synchronization tree and its process definitions. To understand CCS, CSP, and langauges derived from them it is useful to understand the NDISM implied by a set of process definitions.

Consider Figure 15 as a representation of a collection of process definitions. Each node represents a global state of the defined processes perhaps expressed as a tuple of the states of the individual processes if it is desired to examine the composition of the states. Each arc represents a transition from

one state to another via an action or event whose name is the label of the arc. There is one arc label τ that is a place holder naming all *hidden* actions. A hidden action is an action involving portions of the state that are not intended, for the purposes of the definition, to be visible to the user of the system. All other labels are names of actions that the user of the system is to be aware of.

One can take either an active or an inactive view of a system. In the active view, the viewer is a user is operating the system and non-τ labels are names of actions and that can be explicitly chosen or invoked by the user, while τ labels are place holders for internal actions over which the user has no control that cannot be invoked. In the inactive view, the viewer is merely watching the system compute. Non-τ labels name actions that are intended to be visible to the viewer, while τ labels are place holders for internal actions that are intended to be invisible to the viewer.

The literature on CSP and CCS takes both points of view, unfortunately often in the same document. The mixture of viewpoints was confusing to the author of this module, because the reasoning consistent with one point of view is not applicable to the other. Moreover, the passive view of just observing evnts happen puts the viewer in the awkward position of observing unobservable τ events. Therefore, this module takes solely an active point of view.

As with the tree of possible computations of an NDISM, a forking node of a synchronization tree denotes a choice of different actions out of the state represented by the node. If the labels of all of the arcs coming out of a node are distinct and none of them is τ, then the node is said to be *choosable*. In this circumstance each arc has a unique label by which it can be selected and non of them is invisible. Otherwise, if at least two arcs are labelled the same or there is at least one arc labelled τ, then the node is said to be *nonchoosable*. The meaning of being choosable is that in the active view, the external environment can choose which action to take from a state simply by invoking that action's label. If two arcs are labelled the same, choosing that label does not uniquely identify a single arc; thus the choice must be made internally. If at least one arc is labelled τ, then the external environment looses the ability to completely choose because the hidden events denoted by the τ labelled arcs might happen *before* the external environment can exercize a choice by explicitly invoking a non-τ label.

The idea of a node being choosable or nonchoosable simply does not jibe with a passive view. It is simply impossible to passively observe whether or not an active choice has been made when a choice is observed. The set of possible observable events is no different when the choice is purposeful from when it is not!

In the literature on CCS, CSP, and their linguistic descendants, choosable choices are called *deterministic* and nonchoosable choices are called *nondeterministic*. This choice of vocabulary to describe different choices within the specified system is perhaps unfortunate because of the entirely different meaning of these terms at the level of the formal model of the specified system. At the level of the formal model, if the synchronization tree has any forks, the specified system is called nondeterministic, and a deterministic system is one whose synchronization tree has no forks, i.e., at any state there is at most one possible action out of the state. To avoid this confusion, this module continues to use the terms "choosable" and "nonchoosable" for expressing the degree of external control over choices in a nondeterministic system, which has choices.

Each synchronization tree denotes a set of possible computations. In CCS and CSP a computation is a sequence of states for which the actions between them have non-τ labels. In other words, the observed computations consist only of the non-hidden actions that are taken in traversing the synchronization tree from the root towards a leaf. Figure 16 shows the set of computations denoted by the synchronization tree of Figure 15. Some are infinite and some are finite. All start with the initial state from which there was only one possible action, that labelled *a*. Observe that transitions labelled τ and the node at their target are eliminated in building the computation sequences. Thus a computation consists *only* of actions that are externally visible or invokable. In CSP, a computation sequence is called a *trace*.

On this basis, it is easy to see how to construct the tree of possible computations of the NDISM implied by a synchronization tree. Figure 17 shows the extraction of the tree of possible computations from the synchronization tree of Figure 15. The arcs of the synchronization tree are dotted lines and the arcs of the tree of possible computations are solid lines. From each non-τ labelled synchronization tree arc is obtained a tree-of-possible-computations arc that is labelled identically. If the parent of a non-τ labelled synchronization tree arc *a* is labelled τ, then the corresponding arc *a* of the tree of possible computations is extended upward to the parent of the arc labelled τ. This process of extending arcs upward continues recursively until there are no arcs labelled τ. Figure 18 shows the tree of possible computations without the distraction of the synchronization tree from which it was derived.

The two formal systems, CCS and CSP, differ in the ways that forking nodes of synchronization trees can be specified. Each has a slightly different set of choice composition operators. In addition there are slightly different operators for specifying concur-

rency. The effect of these operators is on the representation of the states as a function of the component process states and on the actions that are possible to take from these.

Notation

Choiceless Sequential Processes

The notations to describe the individual processes in CCS and CSP are quite similar and will be described together before continuing on the description of each separately.

The specification of a choiceless sequential process has the flavor of a recursive, grammatical specification of the finite state language of traces whose alphabet is the set of observable event names.

For example, a specification of a vending machine, *VM*, that repeatedly accepts one coin and in response to the coin issues a chocolate is

$VM=coin.choc.VM$ (CCS)
$VM=(coin \rightarrow (choc \rightarrow VM))$ (CSP)

which is to be read "The *VM* accepts a *coin* and *then* issues a *choc* and *then* repeats *itself*".

The set of traces that these specifications generate is

$$\{<coin,choc>^{\infty}\}$$

i.e., the singleton set containing the unbounded sequence of arbitrary repetitions of *coin*, *choc*.

Figure 19 shows the infinite, linear synchronization tree generated by these. Collapsing the synchronization by cycling at the first recurrence point yields the state transition diagram of Figure 20.

In general, if P is a process and a is in the alphabet of P, then

$a.P$ (CCS)
$(a \rightarrow P)$ (CSP)

describes a process that does action a and then does the actions of P.

With the common parts of the two notations described, consideration turns to the choice and concurrency composition operators which differ in the two languages.

Choices

Both notations have ways to express both choosable and nonchoosable choices. Even within a kind of choice, the notations differ in subtle ways. In the following discussions, unless otherwise explicitly stated, P, Q, R, and S are processes and a, b, c, and d are actions, which are assumed to be in the common alphabets of the processes involved. In cases in which what happens when an action is not in the involved processes' alphabets, the alphabets are described explicitly.

In CCS, all choices, choosable and not involve the use of the + operator on processes.

$P+Q$ behaves either as P or as Q. As soon as the first action of one process is performed, the other process is discarded.

A choosable choice is one in which the labels of the first actions of both are not τ and they are distinct, otherwise the choice is nonchoosable. Thus,

$$a.P+b.Q$$

is choosable, but

$a.P+a.Q$,
$a.P+\tau.Q$,
$\tau.P+b.Q$,

and

$$\tau.P+\tau.Q$$

are nonchoosable.

While τ events disappear when considering computations, they cannot always be elided when considering process definitions. Clearly,

$$a.\tau.P=a.P$$

and

$$\tau.\tau.P=\tau.P,$$

but among

$$A=a.A+\tau.b.A$$

and

$$B=a.B+b.B$$

A and B should be different. B may perform either a or b in any state. However, A may reach a state via τ in which b is possible but a is impossible because the other option has been discarded, and there is no way for the environment to control whether or not A gets to this state. Now in a concurrent combination of processes, the environment may offer only an invocation of a, and A is deadlocked. If the the environment offers only an invocation of a tp b, there is no deadlock. Figure 21 shows the state transition diagrams for A and B

In CSP, there is no explicit τ action. Therefore, the distinction between choosable and nonchoosable choices must be made explicit in the operator.

$(a \rightarrow P \mid b \rightarrow Q)$, where $a \neq b$, describes a process which initially does one of a or b. If the chosen action is a, then after that it behaves as P. If the chosen action is b, then after that it behaves as Q.

The "|" operator is not a process operator even though its operands are processes. So, in CSP, one cannot say "$P \mid Q$". Its operands must be of the form "*action→process*" so that there is always an explicit way to choose the desired choice, by selecting its first action. Indeed, the "|" itself is not really the operator; "$(\cdots \rightarrow \cdots \mid \cdots \rightarrow \cdots)$" is the operator taking for operands of types action, process, action,

and process, in that order. The expression $(a \rightarrow P | b \rightarrow Q)$ is considered as a whole much the same way that there are no "if", "then", and "else" operators, and a conditional expression is considered as a whole. Multiway choices are written as flat expressions using one pair of parentheses and more than one "|". Thus, $(a \rightarrow P | b \rightarrow Q | c \rightarrow R)$ is correct and $(a \rightarrow P | (b \rightarrow Q | c \rightarrow R))$ is not. Note also the requirement that the initial actions of each operand of "$(\cdots \rightarrow \cdots | \cdots \rightarrow \cdots)$" be distinct is syntactic. It is syntactically incorrect to write $(a \rightarrow P | a \rightarrow Q)$

Indeed, these syntactic constraints are what distinguish CSP's

$$(a \rightarrow P | b \rightarrow Q)$$

from CCS's

$$a.P + b.Q$$

purposely written without parentheses, which means almost the same thing. The CCS "+" is an operator that can be applied to processes; so $P+Q$ is legitimate. Also it is legitimate and meaningful to write $a.P + a.Q$. The result is a nonchoosable choice; the fact that the first actions of both operands are the same prevents the user from exercizing a choice that chooses between $a.P$ and $a.Q$.

A nonchoosable choice is expressed in CSP with an operator on processes.

$P \lceil\rceil Q$ is a process which behaves either as P or as Q. The choice is made arbitrarily without knowledge or control of the environment.

The other way to obtain a choosable choice in CSP is with a genuine operator on processes.

$P [] Q$ is a process which behaves either as P or as Q. The environment can control which of P or Q is selected provided that the control is exercized on the very first action.

It is clear that if $a \neq b$,

$$(a \rightarrow P) [] (b \rightarrow Q) = (a \rightarrow P | b \rightarrow Q),$$

but

$$(a \rightarrow P) [] (a \rightarrow Q) = (a \rightarrow P) \lceil\rceil (a \rightarrow Q),$$

which in turn equals

$$a \rightarrow (P \lceil\rceil Q).$$

With the choice operators it is possible to define more realistic vending machines. In CSP, a vending machine that issues a chocolate for a dime and caramel for a nickel would be specified as

$$VM = ((dime \rightarrow (choc \rightarrow VM)) | (nickel \rightarrow (caramel \rightarrow VM))).$$

A vending machine that issues either a chocolate or a carmel for each coin, but the customer cannot predict which, would be specified as

$$VM = (coin \rightarrow ((choc \rightarrow VM) \lceil\rceil (caramel \rightarrow VM))).$$

In CCS, the former would be specified as

$$VM = ((dime.choc.VM) + (nickel.caramel.VM)),$$

and the latter would be specified as

$$VM = (coin.((choc.VM) + (caramel.VM))).$$

Figure 22 shows the state transition diagrams of the two newer vending machines.

Concurrency

In both CCS and CSP, concurrency is introduced by operators that cause the individual actions of the argument processes to be interleaved in some way particular to the operator. For some of the operators, there is the possibility of synchronization occurring, that is, of two processes engaging in simultaneous complementing actions. These complementing actions are considered a communication (the "S"es in "CCS" and "CSP"!)

In CCS, there is one concurrency operators on processes of like alphabet, the "|" operator. Before it can be defined, it is necessary to define *complementing* action labels. Two action labels are complementing if they have the same spelling but one has a over bar and the other does not. For example a and $\bar{a}$ are complementing labels.

$P|Q$ is that process whose actions are the interleaved actions of P and Q. In each state, any action from either process is nondeterministically chosen to the be the next action of $P|Q$. The component process whose action is selected is taken to the state that follows the selected actions. If, in both processes the next possible actions are complementing, then one possible next action is the τ action which takes each component process into the state that follows its one of the complementing actions.

If complementing actions are possible, it is possible for the composed process to select one of the complementing actions to take as an individual action.

Given process definitions,

$$A = a.A'$$
$$A' = \bar{c}.A$$

and

$$B = c.B'$$
$$B' = \bar{b}.B.$$

That is,

$A = a.\bar{c}.A$

and

$B = c.\bar{b}.B$.

$A|B$ is definable as

$<A|B> = a.<A'|B>$
$<A|B> = c.<A|B'>$
$<A'|B> = c.<A'|B'>$
$<A'|B> = \bar{c}.<A|B>$
$<A'|B> = \tau.<A|B'>$ (here, $\tau = c\bar{c}$)
$<A|B'> = \bar{b}.<A|B>$
$<A|B'> = a.<A'|B'>$
$<A'|B'> = \bar{b}.<A'|B>$
$<A'|B'> = \bar{c}.<A|B'>$,

where $<P|Q>$ denotes the state of process $A|B$ built from state P of process A and state Q of process B. Figure 23 shows the state transition diagrams of A, B, and $A|B$.

In CSP, there are a variety of operators affecting a concurrent composition of processes. Only the two main ones are discussed here. The distinction between the operators is in their treatment of potential interaction. Recall that in CCS, an interaction *might* occur when two processes are composed and they have complementing actions. If the interaction occurs, the two complementing actions occur and cancel each other to yield a single τ action that is invisible to the observer. Then again, *the interaction* might not occur and each complementing action is left to occur on its own, possibly to interact with another, external complementing action.

In CSP, the *complementing* actions are either like-named events or an input and an output on the same channel. Thus action a on one process complements action a on another process. An output of a value v on channel c is written

$$c!v .$$

This output is complemented by an input to a variable x from the same channel c, which is written

$$c?x .$$

If and when the input-output interaction occurs over channel c, the result is that the value v is assigned to the variable x of the reading process's context.

The simplest concurrent composition operator is the parallel composition operator "||".

If P and Q have the *same* alphabet, then $P||Q$ denotes the process with that same alphabet which behaves as the system composed of P and Q interacting *in lock-step synchronization.*

That is, in each step of $P||Q$, if the next actions of P and Q are complementing, then the common event or the input and output over the common channel happens and each process is taken to its next state via the event or input or output. If the next actions of P and Q are not complementing, i.e., they differ or are not over the same channel, then the processes deadlock.

$(a \rightarrow P)||(a \rightarrow Q) = a \rightarrow (P||Q)$
$(c!v \rightarrow P)||(c?x \rightarrow Q(x)) = c!v \rightarrow (P||Q(v))$

In the latter, the combined event is considered the output to the external environment.

If $a \neq b$ and $c \neq d$, then

$(a \rightarrow P)||(b \rightarrow Q) =$
$(a \rightarrow P)||(c!v \rightarrow Q) =$
$(a \rightarrow P)||(c?x \rightarrow Q(x)) =$
$(c!v \rightarrow P)||(d?x \rightarrow Q(x)) = STOP$

Thus, for two processes with identical alphabets to compose in parallel without deadlock, they must go through sequences of identical actions or input and output over identical channels.

If however, the alphabets of the processes P and Q are different, then events that happen to be in both of their alphabets require simultaneous participation of both processes. However, an event that is in the alphabet of one and not the other is ignored by the other and is done at the one's leisure. The result is that the alphabet of $P||Q$ is the union of the alphabets of P and Q.

Suppose that

a is in the alphabet of P but not in that of Q.
b is in the alphabet of Q but not in that of P.
c and d are in both alphabets, and $c \neq d$.

Clearly $a \neq b$. Then only P can do a, only Q can do b, but P and Q must do c or d simultaneously for $P||Q$ to avoid deadlock. More formally,

$(c \rightarrow P)||(c \rightarrow Q) = c \rightarrow (P||Q)$
$(c \rightarrow P)||(d \rightarrow Q) = STOP$
$(a \rightarrow P)||(c \rightarrow Q) = a \rightarrow (P||(c \rightarrow Q))$
$(c \rightarrow P)||(b \rightarrow Q) = b \rightarrow ((c \rightarrow P)||Q)$
$(a \rightarrow P)||(b \rightarrow Q) = a \rightarrow (P||(b \rightarrow Q))|$
$b \rightarrow ((a \rightarrow P)||Q)$.

Actions in the common alphabet of P and Q require simultaneous participation of both processes, while all remaining actions occur in arbitrary interleaving. To see this, consider the processes P and Q defined

$P = a \rightarrow b \rightarrow c \rightarrow P$
$Q = d \rightarrow c \rightarrow Q$.

The traces of P and Q are shown in Figure 24. Also in that figure is a representation of all possible traces of $P||Q$. Events that lie in a chain of arcs must be ordered as in the chain, but events that lie on opposite paths of parallel chain of arcs are not ordered with respect to each other. The merging of the event arcs for c is supposed to indicate a requirement of simultaneous occurrence. Thus, one possible trace is $a;d;b;c;d;a;b;c; \ldots$.

Note that if P's and Q's alphabets have no actions in common, then P and Q are completely independent, and the traces of $P||Q$ consists of arbitrary interleavings of the individual actions of P and Q. Each such trace is indistinguishable in net effect from even a sequential composition.

Parallel composition interacts with choices in an interesting way. Each operand process is considered to be in the other's external environment. Thus, in any step, when one or both processes offer choosable choices of initial events, only the complementing events, if any, are possible. Each process is, in effect, choosing the other's choosable choices and is choosing the right one to continue the computation. Thus, if

$P=(a\rightarrow b\rightarrow P|b\rightarrow P)$
$Q=(a\rightarrow(b\rightarrow Q|c\rightarrow Q))$

then,

$P||Q=a\rightarrow(b\rightarrow P||(b\rightarrow Q|c\rightarrow Q))=a\rightarrow(b\rightarrow(P||Q))$

In the first step, Q's first action chooses the a choice of P, and in the second step, the $b\rightarrow P$ chooses the b choice of $(b\rightarrow Q|c\rightarrow Q)$. However, letting $P||Q$ be X,

$X=a\rightarrow(b\rightarrow X)$

Note however, if any choice is nonchoosable, the possibility exists that the internal action chooses a first action not complementing the only or chosen action of the choosable choices, yielding a deadlock. As mentioned, the traces generated from $P\lceil\rceil Q$ and $P\,[]\,Q$ cannot distinguish them. However it is possible to put them in a parallel combination with other processes so that $P\lceil\rceil Q$ can deadlock, but $P\,[]\,Q$ cannot. Let $P=(a\rightarrow P)$ and $Q=(b\rightarrow Q)$ where $a\neq b$. For $(P\lceil\rceil Q)\,||\,P$, at every state, the lone P offers a; therefore $P\lceil\rceil Q$ is forced to choose the P alternative because its first action is a. However, if $P\lceil\rceil Q$ is put into a concurrent combination with P, at every state the lone P offers a; however, the $P\lceil\rceil Q$ may choose internally to select the Q alternative, whose first action requires b to proceed. There is no b coming from the other process, so the combination deadlocks.

The other main concurrency operator is the interleave operator "|||" defined for processes with identical alphabets. This operator gets the two processes to execute concurrently with out direct interaction or synchronization.

An action of $P|||Q$ is an action of P or of Q; if both processes are able to execute the same action, then only one is chosen nondeterministically to do that action, leaving it still to be done by the other.

Thus,

$(a\rightarrow P)|||(b\rightarrow Q)=$
$\quad a\rightarrow(P|||(b\rightarrow Q))\,[]\,b\rightarrow((a\rightarrow P)|||Q)$.

If $a=b$ then effectively, the "|||" becomes '$\lceil\rceil$', i.e., the choice becomes nonchoosable. In general the choice is choosable in that if a and b are different, choosing the first action determines which operand is executed. This point is illustrated best when a process is interleaved with another copy of itself. Let $R=(a\rightarrow b\rightarrow R)$. We would expect that $R|||R$ be an arbitrary interleaving of a and b events such that the difference in the numbers of as and bs does not grow without bound.

$R|||R=$
$\quad a\rightarrow((b\rightarrow R)|||R)$
$[\#\ a\rightarrow(R\ |||\ (b\rightarrow R))]$.

Since the first actions are the same for each choice, this

$\quad =a\rightarrow(((b\rightarrow R)|||R)\lceil\rceil(R|||(b\rightarrow R)))$
which
$\quad =a\rightarrow((b\rightarrow R)|||R)$.

However,

$(b\rightarrow R)|||R=(a\rightarrow((b\rightarrow R)|||(b\rightarrow R)))\,[]\,(b\rightarrow(R|||R))$
which
$\quad =(a\rightarrow(b\rightarrow((b\rightarrow R)|||R)))\,[]$
$\quad\ (b\rightarrow(a\rightarrow((b\rightarrow R)|||R)))$.

Letting $(b\rightarrow R)|||R$ be X, we have that

$X=(a\rightarrow b\rightarrow X)\,[]\,(b\rightarrow a\rightarrow X)$, and

letting $R|||R$ be Y, we have that

$Y=(a\rightarrow b\rightarrow Y)\,[]\,(b\rightarrow a\rightarrow X)$.

CSP has a number of other operators for combining processes in concurrent and non-concurrent behavior. These include operators for

- piping the output of the first concurrent process to the input of the second,
- subordinating the second process to the first,
- executing the two processes sequentially,
- interrupting the first process with the second, and
- alternating between the two processes step-by-step.

Observe that CCS's concurrency operator, "|", is different from both of CSP's concurrency operators, "||" and "|||". The "|" allows both synchronization and interleaving, and which one it does at any time is nondeterministic. In CCS,

$(a.P)|(b.Q)=a.(P|(b\ \lceil\rceil))+b.((a.P)|Q)$
$(a.P)|(a.Q)=a.(P|(a.Q))+a.((a.P)|Q)$
$(a.P)|(\bar{a}.Q)=\tau.(P|Q)+a.(P|(\bar{a}.Q))+\bar{a}.((a.P)|Q)$.

This is different from in CSP in which applying a synchronization is not optional. In CSP, synchronization is obligatory with "|", and if it cannot occur, then deadlock happens. Thus,

$(a\rightarrow P)||(b\rightarrow Q)$ is a deadlock

while

$(a \rightarrow P) \| (a \rightarrow Q) = a \rightarrow (P \| Q)$.

On the other hand, in CSP interleaving is obligatory with "|||". Thus,

$(a \rightarrow P) ||| (b \rightarrow Q) =$
$\quad a \rightarrow (P ||| (b \rightarrow Q))$
$[] b \rightarrow ((a \rightarrow P) ||| Q)$.

The CCS concurrency operator is a catch-all, capturing all modes of concurrency, while CSP gives a separate operator to each kind of concurrency. Similarly, CCS choice operator captures all kinds of choice, while CSP gives a separate operator for each kind of choice.

Prevention and Concealment of Actions

CCS has an operator "\" on processes and sets of actions whose effect is to prevent the process from doing any action in the set.

$P\backslash A$ where A is a set of actions is that process obtained from P by preventing P from doing any action in A.

Thus,

$(a.P)\backslash\{a\} = NIL$

is a deadlock because its only possible first step is prevented. The "\" operator can be used to cause a CCS parallel composition to behave like a CSP parallel composition. Recall that

$(a.P) | (\bar{a}.Q) = \tau.(P|Q) + a.(P|(\bar{a}.Q)) + \bar{a}.((a.P)|Q)$.

Preventing the individual a and $\bar{a}$ actions forces their cancellation into the hidden τ event.

$((a.P)|(\bar{a}.Q))\backslash\{a\} = \tau.(P|Q)$,

thus simulating the effect of CSP's $((a \rightarrow P) \| (a \rightarrow Q))$.

CSP has an operator "\" on processes and sets of actions whose effect is to hide as internal actions the actions of the process that are in the set.

$P\backslash A$ where A is a set of actions is that process obtained from P by making all actions in A hidden, internal actions.

Thus,

$(a \rightarrow P)\backslash\{a\} = P\backslash\{a\}$.

Concealment of events can turn a choosable choice into a nonchoosable choice by hiding the actions by which the choice would be chosen.

$(a \rightarrow P) | (b \rightarrow Q)\backslash\{a,b\} = P\backslash\{a,b\} \lceil\rceil Q\backslash\{a,b\}$

CSP's "\" operator can be used to achieve CCS's hiding of cancelling events as τ actions. Suppose $C = \{c!v | v \in$ *alphabet of channel* $c\}$. Then,

$((c!v \rightarrow P) \| (c?x \rightarrow Q(x)))\backslash C = (P \| Q(v))\backslash C$.

CCS does not need a concealment operator because it is implicit in the cancelation of complementing events. CSP does not need a prevention operator because it can be forced by using its obligatory synchronization operator together with processes whose first actions are disjoint.

Example Specification in CSP

The system consisting of the *PRODUCER* and *CONSUMER* running concurrently is

SYSTEM=PRODUCER||CONSUMER .

The definition of *PRODUCER* would be something like

PRODUCER=
(produce_value→
(c!value→PRODUCER)) ,

and the definition of *CONSUMER* would be something like

CONSUMER=
(c?var→(consume_contents_of_var
→CONSUMER)) .

Verification

Hoare's book on CSP, besides being a text book, treats CSP as a formal system and gives algebraic laws that can be used to show that two specifications are equivalent and to prove other properties such as avoidance of deadlock. Indeed as an example of the power of the system, the book devotes considerable space to specification of a solution to the dining philosophers problem and verification of properties of this solution.

Actual Use

CSP has been used to specify a variety of systems. For example, Woodcock has used CSP to specify and prove properties of several different primitives for transaction processing [Woodcock87].

The module author has even seen students use it informally during the course of an informal discussion of how a certain system works. This fact testifies to the naturalness of the notation and the ease with which it is used. Probably its greatest strength is the uninterpreted alphabet that allows events to be considered at any level of detail.

Also in a recent study at SEI comparing several methods, CSP, VDM, and denotational semantics [Place90], to specify concurrent software, CSP was taken as the first one in which to write the specification. While the authors of the report professed to being fair, it seems clear in retrospect that CSP was chosen to be first because working with it involves less notational baggage than other methods; one can write strictly the relevant events.

Tools

At least two tools for writing and verifying CSP specifications have been developed. Both use the trace model of CSP-specified computations as the basis for the formalization embodied in the tool. In one case, Moore has shown how to carry out multilevel decompositions of requirements not unlike those done for the FDM and the HDM (See Sections VI.2 and VI.4.) [Moore90]. He uses the EHDM (See Section VI.4 on HDM) verification system to verify proofs of the correctness of the decomposition, so that properties ascribed to the highest level can be inferred of the lowest level. Camilleri has mechanized the CSP trace model with in Higher Order Logic (HOL) so that the HOL verifyer can be used to prove properties of a CSP specification [Camilleri90].

The Concurrency Workbench is an automated tool for analyzing networks of finite-state processes specified in CCS [Cleaveland89].

Differences Between CCS and CSP

The two languages CCS and CSP clearly have a lot of semantics in common, and their operators are tantalizingly similar, but fundamentally different. Correspondences are shown in Table 1. As can be seen, except for the exact correspondence between CCS's "." and CSP's "→", no CCS operator corresponds to exactly one CSP operator and vice versa.

It seems in retrospect, that CCS is a minimal language providing no concept in more than one operator and even lumping several concepts into one operator. CSP provides more specialized operators each providing only one concept and, in some cases, more than one way to achieve a single concept. For this reason, CSP is probably more useable in writing specifications of concurrent systems. Indeed, one system specification language, LOTOS, which is claimed to be derived from CCS first and CSP second, appears to be more a descendent of CSP than otherwise.

Synchronous CCS

SCCS (Synchronous CCS), also devised by Milner [Cohen86], is an extension of CCS in which a system is viewed as a collection of processes, all running in parallel rather than being interleaved in a nondeterministic fashion. That is, at each step, each process is put through one of its next actions, there being the possibility of choice of next action for a process. Each process is specified to have sequential behavior with choosable or non-choosable choices at any step just as in CCS. The parallel composition operator gets the individual processes working together in lock-step synchrony. Thus in SCCS, a system's state is a tuple of individual process states, and each process is specified by equations much as in CCS.

Synchronization or communication happens when the involved processes do complementing actions in the same step. It is possible for more than two processes to be involved in a synchronization or communication, because all processes are executing in all steps.

In order to permit processes to act at different, varying, and unspecified speeds, the definer of a process can introduce choices of τ actions at any point to allow the other processes to proceed faster with non-τ actions. Of course, it is possible to split lengthy actions into sequences of shorter subactions; the length of the sequence for each split action would be made proportional to an estimated time for the action. Care should be taken in decomposing actions, for it is intended in SCCS that process actions be atomic and non-interruptable.

Milner was able to derive CCS from SCCS. Basically, given an SCCS definition, allow each process to nondeterministically choose a τ action at each steop. Then, in any computation, in each step, have all processes but the one that CCS would choose execute τ actions. The result is a CCS interleaved computation. Moreover, SCCS can express communications involving more than two processes; doing so is impossible in CCS. Therefore, SCCS is strictly more general than CCS.

LOTOS

LOTOS (Language Of Temporal Ordering Specification) [ISO89,Bolognesi87] is a language developed under the auspices of ISO (International Standards Organization) to allow formal specification of OSI (Open Systems Interconnection) computer network architectures and of open, distributed systems in general. LOTOS is based on process algebras and has nothing to do with temporal logic, its name notwithstanding. Officially, LOTOS is based on CCS, but after reading any description of LOTOS, it will be clear that CSP has had greater syntactic and semantic influence.

Besides the basic process algebra to be described below, LOTOS also has features for describing data structures, value expressions, and their types via initial algebraic specifications of abstract data types.

In LOTOS, as in CCS and CSP, a distributed concurrent system is viewed as a *process*. A process may itself be composed of processes, called *sub-processes* to form a hierarchy of process definitions.

A process performs a sequence of *atomic actions*, some of which are *internal* and *unobservable* and others of which interact with other processors forming the *external environment* of the process

Process synchronization is achieved by having several processes execute the same event during the same state transition. A synchronization may involve exchange of data among the processes participating in the event. Notationally and semantically, these synchronization events are as in CSP. They are via like-named events in several processes or via a channel that all processes have access to, with one process doing output and the rest doing input on that channel.

The specification of a process looks like that of a procedure in a programming language. A process has a name and parameters called gates. These gates are the ports by which the communication channels connect to the process. The connection of at least processes with like-named gates forms a channel with that same name. In the body of the process is an expression describing its composition in terms of other sub-processes and events. As an example, consider

process *Max*3[*in*1,*in*2,*in*3,*out*] :=
    hide *mid* **in**
    (*Max*2[*in*1,*in*2,*mid*]
    |[*mid*]|
    *Max*2[*mid*,*in*3,*out*])
    where ...
endproc .

It is assumed that *Max*2 is a process which outputs on its third gate the maximum of the values input on its first two gates. *Max*3 is intended to be a process that outputs on its fourth gate *out*, the maximum of the values input on its first three gates *in*1, *in*2, *in*3. The expression

    (*Max*2[*in*1,*in*2,*mid*]
    |[*mid*]|
    *Max*2[*mid*,*in*3,*out*])

indicates that two instantiations of *Max*2 are to be composed with the output of the first, *mid*, being the first input of the second, via a channel called *mid*. The notation

    |[*mid*]|

is an operator specifying concurrent, i.e., interleaved, execution of its two process arguments, synchronizing via the channel *mid* connection gates named *mid* in the two processes. Note that the two inputs to the first *Max*2 are the first two inputs to *Max*3, the second input to the second *Max*2 is the third input to *Max*3, and the output of the second *Max*2 is the output of *Max*3. The process definition specifies that *mid* is hidden from the external environment of *Max*3.

Associated with this process definition is a diagram that shows the physical composition of the process. For *Max*3 the digram in shown in Figure 25. A process is a box, a channel is a line, a gate is where a channel meets a process. A channel that is entirely within an enclosing box is hidden by the process of that box. A channel that sticks out from a box is connected to an externally visible gate.

Concurrency Operators

In the *Max*3 definition is an example of the most general concurrency operator in LOTOS.

Given processes *P* and *Q* and a set of gates *S*, *P*|*S*|*Q* is able to perform any action at a gate not in *S* or any action that both *P* and *Q* are ready to perform at any gate in *S*.

Note that if, say, *P* is ready to execute an action at an *S* gate and *Q* is not ready to execute an action at the same gate, then *P* must wait until *Q is* ready.

Another of the concurrency operators in LOTOS is the "|||" operator, which carries exactly the meaning of |*S*| with an empty *S*. Since there is no possibility of synchronization, the "|||" operator effectively amounts to specifying interleaved execution, exactly as in CSP.

The third concurrency operator in LOTOS is the "||" operator which is equivalent to |*S*| with *S* being the set of all gates in the system. Thus, the two composed processes are forced to proceed in complete synchrony except for internal events, just as in CSP.

It is interesting that LOTOS defines as its basic parallelism operator something which reduces, as special cases, to the two main concurrency operators of CSP.

There are other operators that are not discussed here in the interest of conserving space.

LOTOS has been used to specify a variety of distributed systems. Moreover there exists automated support for the language, specifically tools to

- write specifications,
- validate and verify specifications, and
- compile specifications to C and Ada.

RAISE

RAISE (Rigorous Approach to Industrial Software Engineering) [Nielsen89] is an attempt to address the problems that prevent the Vienna Development Method (VDM) from being used in large-scale industrial software developments of modern concurrent and distributed systems.

1. VDM is completely manual.
2. The VDM specfication language lacks a satisfactory way to specify concurrency. (This fact kept it from being considered in this module.)
3. The VDM specification language lacks a way to build abstractions and a way to modularize specifications into managemable, separable pieces.

4. The VDM specification language lacks an adequate mathematical semantics.

The RAISE language, methods, and tools are aimed at solving these problems by offering

1. a formally defined, denotational semantics-based specification language with modularization and abstraction building features,
2. a rigorous method for specifying, validating, implmenting, and verifying the correctness of systems, not unlike the methods assumed by FDM (See Section VI.2) and HDM (See Section VI.4), and
3. a collection of tools for editing, printing, checking, and verifying properties of specifications written in the specifcation language.

The RAISE specification language (RSL) is a wide-spectrum language in which the main part dealing with data structures and the nonconcurrent part of alogorithms follows the denotational frame and strongly resembles the VDM specification language. The part of the RSL that deals with concurrency is based on CSP (hence the fact that this discussion is in this section).

10. ASTRAL

ASTRAL [Ghezzi91], developed by Kemmerer and Ghezzi, is an executable specification language for describing real-time systems. Its geneology is instructive. Kemmerer *et al* at the University of California at Santa Barbara developed ASLAN [Auernheimer85] based on Ina Jo in order to provide an executable specification language for roughly the same class of systems specifiable in Ina Jo.

Language

RT-ASLAN [Auernheimer86] is an extension of ASLAN for specifying real-time systems; it was developed by adding timing constraints (in the vernacular sense of the word) to the transforms and constraints (in the formal sense of the word) of ASLAN. A timing expression in a transition specification specifies the time required to execute the transition, i.e., the time limit to which the transform adheres, while that in a constraint specification specifies general time limits to which all transforms are required to adhere. It must be verified that the time limits of the constraint are implied by those of the transforms. One of the strengths of ASLAN and RT-ASLAN is the ability to structure specifications into smaller pieces. They allow layering and composition of specifications, and the formalism allows reasoning about the pieces to be composed into reasoning about the whole specification.

TRIO [Ghezzi90] is a first-order logic language developed at Politecnico di Milano as a formal notation for specifying and verifying timing requirements. It is formally defined, has the type time consisting of numeric values, has the notions of *now*, *future*, and *past*, but lacks features that would make it feasible to use for specifying real-life systems. In particular, it has no modularity and no hierarchical structuring, and its realtime machine-level formal language makes reading specifications extremely difficult for any but the writer of the specifications. However, its formal definition makes it a good basis for a specification and verification environment.

Add to RT-ASLAN features for modeling interprocess communication and specify the resulting language by showing how to translate any of its specifications into TRIO, and you have ASTRAL. From RT-ASLAN, ASTRAL gets the modularity and specification structuring that TRIO lacks, and from TRIO, ASTRAL gets the formal basis it needs for a proof system and specification execution.

The formal model for ASTRAL is a state-machine model like Ina Jo's and ASLAN's. It assumes maximal parallelism with noninterruptable and nonoverlapping transitions in a single process instance. That is, the model behaves as though each process were given its own physical processor and that physical resources, e.g., memory, available to it are unlimited. A processor is never idle when it has a transition available to execute, and if no other transition is pending, a transition is executed as soon as its precondition is satisfied. All variable updates of a transition are treated as taking place in a single atomic action occurring at the end of the transition.

The reason for adopting this model is that it allows the processes to be designed under the assumption of total independence except for explicitly specified communications. Such designs tend to be cleaner than those that deal also with scheduling. After the design is validated, scheduling can be specified separately to insist that timing requirements be met in the face of the reality of limited resources.

Specification and Verification

In ASTRAL, a real-time system is specified by giving a collection of state-machine specifications, one for each type of process, and a single global specification, for the environment in which the process instances sit. Each state machine specification is, in fact, the definition of a process type in the Ada sense, and there may be multiple instantiations of this type in the system as a whole. The assumption mentioned above that the transitions of the processes are nonoverlapping allows properties proved about independent processes to be composed into properties proved about the whole system.

The state variables and transitions defined in a process type specification may be accessed without restriction by any instance of that type. Normally these variables and transitions are not accessible

from outside any containing process. However, variables and transitions may be marked as exported, in which case they are accessible from outside. It appears from the literature about ASTRAL that from outside the defining process, a variable is read-only. Changes to such variables are achieved by invoking exported transitions that do the changes internally. All interprocess communication is via these explicitly exported variables and transitions. This scheme is classic information hiding of process type definitions.

Among the exported variables of any process specification are variables for inquiring the start and ending time for the *k*th last invocation of any other operation with or without specific parameter values in some or all parameter positions. These allow writing of timing expressions that depend on values of variables and on the history of the computation so far.

Whereas RT-ASLAN used interface specifications for interprocess communication, ASTRAL uses a multicast communication model, not unlike the model implemented by an Ethernet or similar network in which each packet is made available to all nodes and a node picks up only packets addressed to it. In the formal model, a message is sent to all processes and only the target process actually gets it. Formally, at the beginning of a transition, the executing process broadcasts the start time. At the end of the transition, the executing process broadcasts both the finish time and the final values of all exported variables for receipt by all processes that have imported them.

The broadcast itself is regarded as taking place instantly. This way, the multicast can be used to model shared memory access as well as communication via a point-to-point channel. In the latter case, the real communications delay must be explicitly specified for an appropriate amount of time. Moreover, the combined effect of universal broadcast of the start and finish times of transition executions and instantaneous broadcast is that timing assertions in all specifications can make use of full information about the time of events without there being any delay to obtain the timing information; that is there is no Heisenberg effect built into the formal model.

An ASTRAL system specification consists of a *global specification* together with a set of individual *specifications* for the process types mentioned as *used* in the global specification. Both kinds of specifications have *type*, *constant*, and *variable* declarations. The variables of any specification make up the state of whatever is described in the specification. As with Ina Jo and ASLAN, a constant or variable may be a function on values or values and variables. Both kinds of specifications may have *invariant*, *constraint*, and *schedule* assertions. An invariant must be true in every state of its containing specification, a constraint must be true of any transition in its containing specification, and a schedule assertion describes the timing of the initiation and termination of the transitions in its containing specifications. Only a nonglobal, process specification may have *transition* specifications, each describing one state transition of a process of the type specified by the specification containing the transition specification. Each transition specification has at least one pair of *entry* and *exit* assertions specifying the state condition that must be true upon a *normal* invocation of the transition and what is true upon completion of such an invocation. A transition specification may also have any number of *exception* and *exit* assertion pairs. Each specifies the state conditions for a so-called exceptional invocation of the transition together with the handling response to that exceptional condition.

Finally, only a process specification may have an *initial* assertion describing the possible initial states of any process of the type specified by the containing specification.

The inovation in ASTRAL is its ability to describe time in its assertions. The basic time primitives are the notion of time as a numeric value over which arithmetic and ordering operators can be applied plus some specific time valued constants and functions such as *now*, *Start(t*, and *End(t)* where *t* denotes an invocation of a transition. Assertions about time can appear in any of the kinds of assertions described above. An example of a transition specification that has timing assertions is:

```
TRANSITION Notify_Death
   ENTRY
      Now - Start(New_Info) ≥
         Timeout
      & ~Channel_Closed
   EXIT
      Msg[Data_Part] = Closed
      & Msg[ID_Part] = Self
      & Channel_Closed
```

This specifies that the transition `Notify_Death` may be invoked only if it has been more than `Timeout` units of time since the start of an invocation of the `New_Info` transition and the channel is not already closed.

A conjunction such as `End(This) - Start(This) ≤ 5 * n` says that the time between the start and end of the current invocation of `This` transition no greater than five times `n` units of time, where `n` might be a parameter of the current transition or a constant, variable, or a function built from all of them. If this conjunct appears in an exit assertion of a transition specification, then the corresponding normal or exceptional invocation is

guaranteed to finish by the specified time. This kind of a conjunct in a schedule assertion in a process or global specification would be subject to proof for the process or globally given what is guaranteed or proved for the contained transitions or processes.

Verification

Verification of an ASTRAL specification follows the same general framework as in Ina Jo. The specification is compiled into conjectures written in the TRIO language. These conjectures assert the following:

1. In each process specification,
   - the invariant is implied by the initial assertion,
   - the invariant is preserved by each transition,
   - the constraint is implied by each transition; and
2. in the global specification,
   - the invariant is implied by each process's invariant,
   - the constraint is implied by each process's constraint,
   - the schedule requirement is implied by each process's schedule.

In the above, when something is implied by a transition, it is implied by its entry and exit assertions pair and by each of its except and exit assertions pair.

Tools

The authors realized that a interpretable specification language without an interpreter is not very useful; therefore the design of the language proceeded in parallel with the design of the environment for compiling ASTRAL specifications into TRIO specifications and the TRIO evaluator. At the present time, a syntax-directed ASTRAL editor exists in prototype form. The translation from ASTRAL to TRIO is done by hand, but a syntax-directed translator is under development. Finally, a TRIO symbolic executor, written in PROLOG, is available for distribution.

Actual Use

ASTRAL appears to this author as the first real-time system specification language that has a chance of being applied to the design specification and verification of real-time systems in a way that whether the real-time constraints are met are subject to formal mathematical verification. ASTRAL is too experimental to have had any industrial strength application.

VII. Current Status

Of the above described specification and verification environments, at least AFFIRM and HDM have undergone enhancement to yield more powerful specification languages and verification environments. AFFIRM has been upgraded to AFFIRM-85 [Musser85a, Musser85b]. AFFIRM-85 has more facilities for structuring specifications hierarchically, and its environment has a library of reuseable proofs.

The language of Enhanced HDM [Rushby91a] is Revised SPECIAL [Levitt85]. It has a number of improvements over SPECIAL, including parameterizable modules, the use of second order predicate calculus, and the treatment of program operations as data objects. A more powerful logic is needed to support these enhancements [Shostak82]. More recently, EHDM has been used successfully to help construct a verifiably correct distributed clock synchronization algorithm [Rushby91b, Rushby91c].

Gypsy has been ported to several other machines including Symbolics machines [Smith85].

Another approach that has been explored is that of symbolic execution of an operational model specification, such as AFFIRM, Ina Jo, or VDL. UNISEX [Kemmerer83, Kemmerer85] is a successful implementation of this idea for non-concurrent programs written in Pascal. Inatest has been developed to allow symbolic execution of Ina Jo specifications in order to test whether what is desired equals what is specified [Eckmann85]. Others are exploring extending this idea to deal with Ada tasking [Dillon 88a, Dillon88b, Harrison88].

Glossary

asynchronous execution
: the appearance of parallel execution achieved through true parallel execution or multiplexing, in which what happens next is unpredictable, and in which the concept of next may not even be meaningful

bipartite graph
: an ordered graph with two disjoint sets of nodes, such that for each arc, the head and tail nodes are in the opposite sets

concurrent program
: a program whose execution consists of or gives rise to asynchronous execution

deadlock
: the situuation that occurs when all nonterminated processes in a computation are asleep;

i.e., each is waiting for a resource that another provides; an alternative definition is when at least one process is waiting for an event that cannot occur

distributed computing

a collection of processors not sharing memory, but sharing communication channels that allow the processes on them to communicate with each other

fairness

a property of a scheduling algorithm that insures that all awake processes eventually get to run; if a scheduler is fair then one method of starvation cannot happen

interference

the act of two or more processors attempting and possibly succeeding to use the same physical device (including memory locations) at the same time

multiplexing

(IEEE:multitasking) a mode of operation in which two or more tasks are executed in an interleaved manner (by a single processor)

multiprocessing

a mode of operation in which two or more processes are executed simultaneously (IEEE uses "concurrently") by separate processing units that have access to a common main storage

nondeterminate

a nondeterministic computation is nondeterminate if the differing orders of computation cause different final results

nondeterministic

a computation is nondeterministic if at each state, the step to be done next is not defined by the program being computed; the choice of what is to be done next is made by an agent external to the program

parallel execution

simultaneous execution of more than one process, which can happen either in multiprocessing or distributed computing

partial correctness

a form of program correctness characterized by the program meeting its requirements in all of its halting computations; a program is partially correct if for all input that meets its input requirements, the program meets its output requirements whenever it halts

process item

a data structure keeping the state of a process in the memory accessible to the processors which can execute on behalf of the process

process

an execution of a program or a portion thereof

process status

at the level of the program, a process can be either awake, asleep, or terminated; a process is awake if it is is capable of running and would run if there were sufficient processors available to run it; a process is asleep if it cannot run because of a condition defined in the program it is executing, i.e., it is waiting for a resource from another process with which it shares data; a process is terminated if it cannot run any more, e.g., it has finished its work or it has committed an unrecoverable error; at the level of the implementation, an awake process is either running or ready; it is running if a processor is running it; it is ready if at the moment no processor is available to run it; note the distinction between ready and asleep processes; the latter could not be run even if there were enough processors available

processor

an agent capable of executing a program or portion thereof

program

(IEEE:computer program) a combination of computer instructions and data definitions that enable computer hardware to perform computational or control functions

proper termination

if and when a computation terminates, i.e., there are no awake processes, then there are no asleep processes still waiting for a resource; proper termination implies absence of deadlock.

race condition

the situuation that occurs when the resolution of interference for a particular device can cause unpredictable results

real time

(IEEE: real time) pertaining to a system or mode of operation in which computation is performed during the actual time that an external process occurs, in order that the computation results can be used to control, monitor, or respond in a timely manner to the external process

starvation

the situuation that occurs when an awake process fails to get any work done because it is denied a resource it needs; either it is busy waiting for a resource being held on to by another process or it is being denied a processor by the processor scheduling algorithm

synchronous execution

the absence of asynchronous execution, in which what happens next is completely predictable

task

the same as **process**

total correctness

a form of program correctness characterized by the program halting for all inputs and meeting its requirements in all cases; a program is totally correct if for all input that meets its input requirements, the program halts and meets its output requirements

Teaching Considerations

Suggested Schedules

There is far more material in this module than can be covered in one semester or one quarter. Therefore, the instructor will have to pick and choose from the topics mentioned in the module.

Since the emphasis of the course should be the application of the formal methods of specification and verification to the development of concurrent software, the instructor should pick topics and methods that he or she is most comfortable working with in front of the class. The instructor will have to personally demonstrate the methods in front of the student and will have to be prepared to field difficult questions from the students and bug emergencies as the students find errors in what the instructor is presenting.

The outline below represents how the module author would teach the course based on his own preferences and familiarities.

1. Foundational Background (Section I)
2. Ada Concurrency Features (Material from the module [Feldman90])
3. Properties of Concurrent Programs (Section II)
4. Operational Models (Section III.1)
5. Temporal Logic (Section III.3)
6. Specifications and Verifications of Protocols in Operational and Temporal Models (Sections IV.3 and V)
7. Use of FDM and SARA to do these specifications and verifications (Sections VI.3 and VI.7)

Of course, the instructor who wishes to follow a similar outline but is not comfortable with the specific languages, methods, and tools, could use others, e.g., Concurrent Pascal, Axiomatic Semantics, Operating System Security, and AFFIRM.

The module author considers it critical to cover Topics 1, foundational background, and 2, some language's concurrency features. The students need this material to make what follows an abstraction of concepts they already know rather than totally new material from which they may derive an understanding.

Worked Examples

The worked examples should be sufficiently complex that the act of specifying them actually teaches the students about the function of the software. Protocols are ideal for this because they are generally difficult to understand without some model. For example, the students might be shown in lectures formal definitions of the Alternating Bit protocol in each of the languages and notations covered.

Exercises

If the class goals include the ability to write specifications of concurrent systems, then in all probability, the instructor will be writing some specifications in front of the students in class. Then, as homework, the students should be asked to define another more complex protocol, say, Stenning's Data Transfer Protocol [Sunshine82], in each of the same languages and notations and to prove implementation of the basic service protocol, safety, and liveness properties.

It would be useful to get a copy of or get access to one of the formal specification and verification environments so that students can attempt to use these in which to carry out their assigned specifications and verifications.

From where can these environments be obtained? Below is information on how to obtain environments for most of the languages covered in detail in Section VI. Note that although the module author has seen some of these environments in operation, he has never tried porting any of them from its original site.

PAISLey

As mentioned earlier, the PAISLey environment, including the processor and other tools, is sold through AT&T's UNIX toolchest. Use the UNIX toolchest, dial the computer at 201-522-6900 and log in as `guest`. The system will prompt you. The environment is also available free to any academic institution. Faculty members interested in ordering it should send to Pamela Zave a request for the environment on institutional letterhead. Her address is Dr. Pamela Zave, Software and Systems Research

Center, AT&T Bell Laboratories, Room 3D-426, Murray Hill, NH 07974, U.S.A., e-mail: pamela@allegra.att.com.

AFFIRM

Work has ceased on AFFIRM entirely. The original AFFIRM, developed at ISI, runs on only the PDP 10 class machines, and there appear to be none around except in museums.

According to David Musser, the leader of the project to develop AFFIRM-85, Affirm-85, a version of Affirm available from Rensselaer Polytechnic Institute, runs on VAX/VMS machines. No license for Affirm-85 is required, *and no support is provided*, but licenses are required for Interlisp-VAX from DEC (at least a run-time license) and for Unipress Emacs (which is used as part of the user interface) from Unipress, Inc. Affirm-85 was developed at General Electric Corporate Research and Development Center (as a porting and extension of the original Affirm system developed at USC/Information Sciences Institute) but is in the public domain, as it was sponsored by the U.S. Air Force. It may also be possible to get Affirm-85 from Rome Air Development Center, Griffiths Air Force Base, Rome, NY (John Faust). The distribution tape from Rensselaer includes the source files and a binary executable file. Documentation describing installation procedures and differences from the original ISI Affirm system is included (copies of the original Affirm documentation may also be obtained from Rensselaer, or from ISI). A nominal charge will be made for tape copying and document reproduction. Send inquiries to Professor David Musser, Rensselaer Polytechnic Institute, Computer Science Department, Amos Eaton Hall, Troy, NY 12180, U.S.A., e-mail: musser@turing.cs.rpi.edu.

GYPSY VERIFICATION ENVIRONMENT (GVE)

The GVE is available for distribution subject to certain export restrictions imposed by the U.S. government. The GVE is a free product though there are some conditions for release. Requests can be sent to Ron Olphie, Computational Logic, Inc., 1717 W. 6th Street, Suite 290, Austin, TX 78703, U.S.A., email: olphie@cli.com 512-322-9941.

Various system documentation, including a user's manual, is available. These consist mainly of Computational Logic, Inc. technical reports. Contact Ron Olphie at the address above to obtain this documentation.

FDM

The FDM environment for Sun 3's is available at no cost for as long as the cost continues to be covered by National Computer Security Center. Please contact Deborah Cooper, Unisys Corp., Formal Methods 5731 Slauson Ave., Culver City, CA 90230, 213-338-3727, e-mail:cooper@culv.unisys.com for details. Available with the distribution is a full set of user documentation, including the FDM User Guide with a complete tutorial example. Unisys also offers a two-week Advanced FDM Course emphasizing hands-on experience. This course includes reviews of first order logic and state machines. There is a plan to develop a Basic Course, as a prerequisite to the Advanced course, for brand new users.

Unisys has recently developed the Model Executor as a simulator of Ina Jo specifications. It has proved to be a good pedagogical tool for understanding the semantics of Ina Jo and Ina Jo specifications. It is written in Quintus Prolog. Therefore, a site not already licensed for Quintus Prolog must pay a $400.00 licensing fee to run the Model Executor. The Model Executor is accompanied by documentation, *Model Executor User Guide*.

HDM

The original HDM no longer exists. There is an newer version of the method, called EHDM. EHDM has been developed by the Computer Science Laboratory (CSL) of SRI International for the U.S. Government and is therefore in the Public Domain. SRI International has no wish to restrict the availability of EHDM, but distribution of the EHDM system and its documentation is subject to controls imposed by the U.S. Government. Permission to obtain copies of the EHDM system is generally routine for Agencies of the U.S. Government, and for U.S. Corporations working on U.S. Government contracts. Policies on wider distribution are unclear at present.

Those interested in using EHDM should contact John Rushby at the Computer Science Laboratory, SRI International, 333 Ravenswood Avenue, Menlo Park, CA 94025 USA, 415-859-5456, FAX: 415-859-2844, e-mail: rushby@csl.sri.com.

EHDM Version 5.1.4 is currently available for Sun-3 and Sun-4 workstations (Symbolics machines are no longer supported). At least 8 MB of real memory (16 MB recommended), 35 MB of swap space (50 MB recommended), and about 32 MBytes of file space are required. EHDM is implemented in Sun Common Lisp and you will need the appropriate RTU license from Sun. A full GNU Emacs is needed since EHDM uses this as its interface.

The documentation available includes a Language manual, a System manual, a Formal Semantics, and

a rather good Tutorial. These are all available the same restrictions mentioned above.

SARA

The old SARA system is not available and the latest CoSARA (Cooperative SARA) is not ready for distribution. CoSARA will allow several designers to be working on the same design at the same time from different workstations.

P-NUT

P-NUT is distributed as part of the standard Arcadia distribution tape. There is a licensing agreement that must be signed, and there is a nominal distribution fee. For further information about the Arcadia distribution, please contact Professor Richard Taylor, Information & Computer Science Department, University of California, Irvine, CA 92717, U.S.A. e-mail: taylor@ics.uci.edu.

STATEMATE

STATEMATE may be licensed on a commercial basis from i-Logix, Inc., 22 Third Avenue, Burlington, MA 01803, U.S.A., 617-272-8090, FAX: 617-272-8035. It is available on a number of hardware platforms including VAX/VMS. It comes with a graphic user interface for constructing the models and an analyzer for analyzing and simulating models.

Caveats

Beware of typographical errors in textbooks adopted for any course. While you, the instructor, can weather these errors, the students may not be able to distinguish between their own misunderstanding and genuine errors in the text.

Bibliography

Ambler78

Ambler, A.L., D.I. Good, J.C. Browne, W.F. Burger, R.M. Cohen, C.G. Hoch, and R.E. Wells. "Gypsy: A Language for Specification and Implementation of Verifiable Programs." *Proceedings of the ACM Conference on Language Design for Reliable Software.* New York: ACM, 1978, 1-10.

> ***Abstract:*** *An introduction to the Gypsy programming and specification language is given. Gypsy is a high-level programming language with facilities for general programming and also for system programming that is oriented toward communications processing. This includes facilities for concurrent processes and process synchronization. Gypsy also contains facilities for detecting and processing errors that are due to the actual running of the program in an imperfect environment. The specification facilities give a precise way of expressing the desired properties of the Gypsy programs. All of the features of Gypsy are fully verifiable, either by formal proof or by validation at run time. An overview of the language design and a detailed example program are given.*

Apt80

Apt, K., N. Francez, and W.P. de Roever. "A Proof System for Communicating Sequential Processes." *ACM Trans. Prog. Lang. and Syst. 2,* 3 (July 1980), 359-385.

> ***Abstract:*** *An axiomatic proof system is presented for proving partial correctness and absence of deadlock (and failure) of communicating sequential processes. The key (meta) rule introduces cooperation between proofs, a new concept needed to deal with proofs about synchronization by message passing. CSP's new convention for distributed termination of loops is dealt with. Applications of the method involve correctness proofs for two algorithms, one for distributed partitioning of sets, the other for distributed computation of the greatest common divisor of* n *numbers.*

Auernheimer85

Auernheimer, B. and R.A. Kemmerer. *ASLAN User's Manual.* Report No. TRCS84-10, Department of Computer Science, University of California, Santa Barbara, CA, 1984.

> This document serves as a guide to the ASLAN specification language and use of the ASLAN language processor. It introduces the strategies underlying the ASLAN approach to system specification, and explains the ASLAN approach towards correctness and consistency conjectures. Detailed examples of the syntax and semantics of the ASLAN language are presented and a nontrivial, syntactically correct specification example and associated correctness conjectures are included.

Auernheimer86

Auernheimer, B. and R.A. Kemmerer. "RT-ASLAN: A Specification Language for Real-Time Systems." *IEEE Trans. Software Eng. SE-12,* 9 (1986), 879-889.

> ***Abstract:*** *RT-ASLAN is a formal language for specifying real-time systems. It is an extension of the ASLAN specification language for sequential systems. Some of the features of the ASLAN language, such as constructs for writing procedural semantics in a nonprocedural logical language, are highlighted. The RT-ASLAN language supports specification of parallel real-time processes through arbitrary levels of abstraction; processes do not have to be specified to the same level of detail. Communicating processes use an interface process as an abstract data type representing shared information. From RT-ASLAN specifications, performance correctness conjectures are generated. These conjectures are logic statements whose proof guarantees the specification meets critical time bounds. A detailed example as well as a discussion of the advantages and disadvantages of formal specification and verification are included.*

Bach86

Bach, M.J. *The Design of the UNIX Operating System.* Englewood Cliffs, NJ: Prentice-Hall, 1986.

> This book describes the data structures and algorithms used to implement System V.

Balzer81

Balzer, R. "Transformational Implementation: An Example." *IEEE Trans. Software Eng. SE-7* (1981), 3-14.

> ***Abstract:*** *A system for mechanically transforming formal program specifications into efficient implementations under interactive user control is described and illustrated through a detailed example. The potential benefits and problems of this approach to software implementation are discussed.*

Balzer85
Balzer, R. "A 15 Year Perspective On Automatic Programming." *IEEE Trans. Software Eng. SE-11* (1985), 1257-1268.

Abstract: Automatic programming consists not only of an automatic compiler, but also some means of acquiring the high-level specification to be compiled, some means of determining that it is the intended specification, and some (interactive) means of translating this high-level specification into a lower-level one which can be automatically compiled.

We have been working on this extended automatic programming problem for nearly 15 years, and this paper presents our perspective to this problem and justifies it in terms of our successes and failures. Much of our recent work centers on an operational testbed incorporating usable aspects of this technology. This testbed is being used as a prototyping vehicle for our own research and will soon be released to the research community as a framework for development and evolution of Common Lisp systems.

Barringer85
Barringer, H. *A Survey of Verification Techniques for Parallel Programs.* Lecture Notes in Computer Science, no. 191. Berlin: Springer-Verlag, 1985.

This book is a first attempt at trying to document some of the many verification methods for parallel and distributed software. Among the included papers are: Flon & Suzuki: Total Correctness of Parallel Programs, Jones: Development of Interfering Programs, Lamport: Verification of Concurrent Programs, Owicki & Gries: Verification of Parallel Programs, and methods for message-based parallelism, Apt & Francez & De Roever: Verification of CSP, Barringer & Mearns: Verification of Ada Tasks, Levin & Gries: Verification of CSP, Misra & Chandy: Proofs of Process Networks, and Zhou & Hoare: Correctness of Communicating Processes.

Barton88
Barton, A., J. Gingerich, S. Holtsberg, G. Smith, and D.J. van Buer. *FDM Testing Tools User Guide.* TM-8446/003/01, Unisys Corporation, Culver City, CA, Oct. 1989.

(From Preface) This guide is for maintainers and testers of the Unisys Formal Development Methodology (FDM). It describes the procedures and tools used for testing the tools of the FDM: the Ina Jo processor, the ITP, and Short.

Bates81
R.L. Bates and S.L. Gerhart. *AFFIRM Annotated Transcripts.* USC Information Sciences Institute, Marina del Rey, CA, Feb. 1981.

(From Preface) The *AFFIRM* Annotated Transcripts volume illustrates a number of features of *AFFIRM*. Each transcript is prefaced with a short description of what the transcript deals with and other highlights. All of these transcripts are from the current system as of the writing of this volume. Some of the proofs are highly polished and many people contributed to them.

Bekic74
Bekic, H., D. Bjorner, W. Henhapl, C.B. Jones, and P. Lucas. *A Formal Definition of a PL/I Subset, Parts I and II.* Technical Report 25.139, IBM Laboratory, Vienna, 1974.

This document represents the first use of the emerging Vienna Definition Method (VDM) specification language to specify the run-time semantics of the programming language PL/I.

Bell73
Bell, D.E. and L.J. LaPadula. *Secure Computer Systems: Mathematical Foundations and Model.* ESD-TR-73-278, Volumes 1 and 2, The MITRE Corporation, Bedford, MA, 1973.

This paper defines the basic model of data security that almost everyone uses to specify system security. It has the concept of subjects (processes) of different levels of clearance having access to objects (files) of different levels of sensitivity.

Ben-Ari81
Ben-Ari, M., Z. Manna, and A. Pnueli. "The Temporal Logic of Branching Time." *Conference Record of Eighth Annual ACM Symposium on Principles of Programming Languages.* New York: ACM, Jan. 1981, 164-176.

Abstract: A temporal language and system are presented which are based on branching time structure. By the introduction of symmetrically dual sets of temporal operators, it is possible to discuss properties which hold either along one path or along all paths. Consequently it is possible to express in this system all the properties that were previously expressible in linear time or branching time systems. We present an exponential decision procedure for satisfiability in the language based on tableaux methods, and a complete deduction system. An associated temporal semantics is illustrated for both structured and graph representations of programs.

Benzel84
Benzel, T. "Further Analysis of the SCOMP System Verification." *Seventh Dod/NBS Security Conference.* Arlington, Virginia: DoD Computer Security Evaluation Center, Sept. 1984.

This paper is one in a series that examines the formal verification process used to certify the SCOMP operating system. This is a long and stringent mathematical process that is required to prove formally the protection properties of a system. It is noteworthy that SCOMP is certified A1 under the Orange Book criterion by the US DoD.

Berg82
Berg, H.K., W.E. Boebert, W.R. Franta, and T.G. Moher. *Formal Methods of Program Verification and Specification.* Englewood Cliffs, NJ: Prentice-Hall, 1982.

The book has a wide-ranging survey of verification. Chapter 6 deals with correctness of parallel programs and surveys several methods of dealing with shared-memory parallelism and interference. The bibliography is extensive with 166 entries.

Bernstein87
Bernstein, P.A., V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Reading, MA: Addison-Wesley, 1987.

This book has a tutorial explanation of two-phase commit.

Berry72
Berry, D.M. "The Equivalence of Models of Tasking." *ACM SIGPLAN Notices 7,* 1 (Jan. 1972), 170-190.

Abstract: *A technique for proving the equivalence of implementations of multi-tasking programming languages is developed and applied to proving the equivalence of the contour model and a multi-tasking version of the copy rule.*

Berry85
Berry, D.M. "A Denotational Semantics for Shared-Memory Parallelism and Nondeterminism." *Acta Informatica 21* (1985), 599-627.

Abstract: *It is first shown how to construct a continuation from a deterministic Vienna Definition Language control tree. This construction is then applied to nondeterministic control trees. The result is a denotational but not quite continuation semantics for arbitrary shared-memory nondeterminism and parallelism. The implications of this result are discussed.*

Berztiss87
Berztiss, A. *Formal Specification of Software.* Curriculum Module SEI-CM-8, DTIC: ADA 236362, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa., Oct. 1987.

(From Capsule Description) This module introduces methods for the formal specification of programs and large software systems, and reviews the domains of application of these methods. Its emphasis is on the functional properties of software. It does not deal with the specification of programming languages, the specification of user-computer interfaces, or the verification of programs. Neither does it attempt to cover the specification of distributed systems.

Berztiss88
Berztiss, A. and M.A. Ardis. *Formal Verification of Programs.* Curriculum Module SEI-CM-20, DTIC: ADA 235775, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa., Dec. 1988.

(From Capsule Description) This module introduces formal verification of programs. It deals primarily with proofs of sequential programs, but also with consistency proofs for data types and deduction of particular behaviors of programs from their specifications. Two approaches are considered: verification after implementation that a program is consistent with its specification, and parallel development of a program and its specification. An assessment of formal verification is provided.

Bevier87
Bevier, W.R. *A Verified Operating System Kernel.* Technical Report 11, Computational Logic, Inc., Austin, TX, 1987.

Abstract: *We present a multitasking operating system kernel, called KIT, written in the machine language of a uni-processor von Neumann computer. The kernel is proved to implement, on this shared computer, a fixed number of conceptually distributed communicating processes. In addition to implementing processes, the kernel provides the following verified services: process scheduling, error handling, message passing, and an interface to asychchronous devices. The problem is stated in the Boyer-Moore logic and the proof is mechanically checked with the Boyer-Moore theorem prover.*

Bolognesi87
Bolognesi, T. and E. Brinksma. "Introduction to the ISO Specification Language LOTOS." *Computer Networks and ISDN Systems 14* (1987), 25-59.

Abstract: *LOTOS is a specification language that*

has been specifically developed for the formal description of the OSI (Open Systems Interconnection) architecture, although it is applicable to distributed, concurrent systems in general. In LOTOS, a system is seen as a set of processes which interact and exchange data with each other and their environment. LOTOS is expected to become an ISO standard by 1988.

Britton84

Britton, D.E. "Formal Verification of a Secure Network with End-to-End Encryption." *Proceedings of the 1984 Symposium on Security and Privacy.* Washington, DC: IEEE Computer Society, April 1984.

Abstract: *A formal specification and verification of a simple secure communications network using end-to-end encryption is presented. It is shown that all data sent over the network is encrypted and all hosts on the network exchange messages only if they are authorized to do so. The network and its hosts are modelled by a set of concurrent processes that communicate via unidirectional buffers. Each process is viewed as a state machine. The specification has been formally verified using the commercially available VERUS verification system.*

Bustard90

Bustard, D.W. *Concepts of Concurrent Programming.* Curriculum Module SEI-CM-24, DTIC: ADA 223897, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa., April 1990.

(From Capsule Description) A concurrent program is one defining actions that may be performed simultaneously. This module discusses the nature of such programs and provides an overview of the means by which they may be constructed and executed. Emphasis is given to the terminology used in this field and the underlying concepts involved.

Camilleri90

Camilleri, A.J. "Mechanizing CSP Trace Theory in Higher Order Logic." *IEEE Trans. Software Eng. 16,* 3 (Sept. 1990), 993-1004.

Abstract: *The process algebra CSP is widely used for formal reasoning in the areas of concurrency, communication, and distributed systems. Mathematical proof plays a key role in CSP reasoning, but despite this, little mechanical proof support has been developed for CSP to facilitate the exercise and eliminate the risk of human error. In this paper we described how a mechanized tool for reasoning about CSP can be developed by customizing an existing general-purpose theorem prover based on higher order logic. We investigate how the trace semantics of CSP operators can be mechanized in higher order logic, and show how the laws associated with these operators can be proved from their semantic definitions. The resulting system is one in which natural-deduction style proofs can be conducted using the standard CSP laws.*

Chandy88

Chandy, K.M. and J. Misra. *Parallel Program Design.* Reading, MA: Addison-Wesley, 1988.

This book defines the UNITY parallel program design language and and develops a complete theory of programming with UNITY and of the meaning of UNITY programs.

Cheheyl81

Cheheyl, M.H., M. Gasser, G.A. Huff, and J.K. Millen. "Verifying Security." *ACM Computing Surveys 13,* 3 (Sept. 1981), 279-340.

Abstract: *Four automated specification and verification environments are surveyed and compared: HDM, FDM, Gypsy, and AFFIRM. The emphasis of the comparison is on the way these systems could be used to prove security properties of an operating system design.*

The acronyms HDM and FDM stand for Hierarchical Development Methodology and Formal Development Methodology, respectively. HDM is based on the specification language SPECIAL, and the (non-interactive) Boyer-Moore theorem prover. FDM makes use of the Ina Jo specification language and an interactive theorem prover. Gypsy is a highly integrated environment intended for the incremental verification of software. In AFFIRM, software development is regarded as the specification and implementation of abstract data types, and specifications are written as algebraic axioms. Although this survey deals specifically with the verification of security, it provides clear descriptions of the four verification methodologies listed above and is an invaluable guide to further reading. The largest application example known at the time is indicated for each environment.

Chen83

Chen, B.S. and R.T. Yeh. "Formal Specification and Verification of Distributed Systems." *IEEE Trans. Software Eng. SE-9,* 6 (Nov. 1983), 710-722.

Abstract: *Computations of distributed systems are extremely difficult to specify and verify using traditional techniques because the systems are inherently concurrent, asynchronous, and nondeterministic. Furthermore, computing nodes in a distributed system may be highly independent of each other, and the entire system may lack an accurate global clock.*

In this paper, we develop an event-based model to specify formally the behavior (the external view) and the structure (the internal view) of distributed systems. Both control-related and data-related properties of distributed systems are specified using two fundamental relationships among events: the "precedes" relation, representing time order; and the "enables" relations, representing causality. No assumption about the existence of a global clock is made in the specifications.

The specification technique has a rather wide range of applications. Examples from different classes of distributed systems, include communication systems, process control systems, and a distributed prime number generator, are used to demonstrated the power of the technique.

The correctness of a design can be proved before implementation by checking the consistency between the behavior specification and the structure specification of a system. Both safety and liveness properties can be specified and verified. Furthermore, since the specification technique defines the orthogonal properties of a system separately, each of them can be verified independently. Thus, the proof technique avoids the exponential state-explosion problem found in state-machine specification techniques.

Cleaveland89

Cleaveland, R., J. Parrow, and B. Steffen. *The Concurrency Workbench: a Semantics Based Tool for the Verification of Concurrent Systems.* LFCS report series, ECS-LFCS-89-83, Dept. of Computer Science, University of Edinburgh, Edinburgh, UK, 1989.

Abstract: *The Concurrency Workbench is an automated tool that caters for the analysis of networks of finite-state processes expressed in Milner's Calculus of Communicating Systems. Its key feature is its scope: a variety of different verification methods, including equivalence checking, preorder checking, and model checking, are supported for several different process semantics. One experience from our work is that a large number of interesting verification methods can be formulated as combinations of a smaller number of primitive algorithms. The Workbench has been applied to examples involving the verification of communications protocols and mutual exclusion algorithms and has proven a valuable aid in teaching and research.*

Cohen86

Cohen, B., W.T. Harwood, and M.I. Jackson. *The Specification of Complex Systems.* Wokingham, England: Addison-Wesley, 1986.

This brief, 143 page book explores some aspects of the electronic office by means of equational algebraic specification and the Vienna Development Method (VDM). Chapter 6 covers specification of concurrent systems. Chapter 7 describes formal methods in the development environment. This chapter includes a listing of centers of current research activity in the more theoretical approaches to specification, and of experimental specification languages.

Comer84

Comer, D. *Operating System Design: the Xinu Approach.* Englewood Cliffs, NJ: Prentice-Hall, 1984.

This book describes the Xinu system, similar to the UNIX kernel, that runs on the LSI-11 minicomputer. The book describes and gives the complete C code for the system.

Cousot82

Cousot, P. and R. Cousot. "Induction Principles for Proving Invariance Properties of Programs." In *Tools and Notions for Program Construction,* D. Neel, ed. Cambridge, England: Cambridge University Press, 1982, 75-119.

Abstract: *We propose sixteen sound and complete induction principles for proving program invariance properties. We study their relationships and show that they can be derived from each other by commuting mathematical transformations. Only five of these induction principles correspond to already known invariance proof methods. We choose a non-conventional induction principle and construct corresponding partial correctness, nontermination and clean behavior proof methods. When constructing these new proof methods, we informally apply our mathematical approach published earlier. This essentially consists in decomposing the global inductive invariant involved in the induction principle into an equivalent set of local invariants and in deriving the corresponding verification condition.*

Dennis68

Dennis, J.B. "Programming Generality, Parallelism, and Computer Architecture." In *Information Processing 68.* Amsterdam: North-Holland, 1968, 484-492.

Abstract: *Parallelism and programming generality are increasingly important attributes of computer systems. Yet their joint influence on computer architecture has not been felt. In this paper, a program graph description of algorithms is developed that meets the requirements of programming generality and allows asynchronous parallel execution without loss of determinism. A machine organization inspired by the program graph models is sketched.*

Dijkstra68
Dijkstra, E.W. "The Structure of 'THE' Multiprogramming System." *Comm. ACM 11*, 5 (May 1968), 341-346.

Abstract: *A multiprogramming system is described in which all activities are divided over a number of sequential processes. These sequential processes are placed at various hierarchical levels, in each of which one or more independent abstractions have been implemented. The hierarchical structure proved to be vital for the verification of the logical soundness of the design and the correctness of its implementation.*

Dijkstra76
Dijkstra, E.W. *A Discipline of Programming.* Englewood Cliffs, NJ: Prentice-Hall, 1976.

The topic of weakest preconditions is developed by the master himself. The book has no index and no bibliography.

Dillon88a
L.K. Dillon, R.A. Kemmerer, and L.J. Harrison. *An Experience with Two Symbolic Execution-Based Approaches to Formal Formal Verification of Ada Tasking Programs.* TRCS88-6, Department of Computer Science, University of California, Santa Barbara, CA, 1988.

Abstract: *There have been several efforts to use symbolic execution to test and analyze concurrent programs. Recently proof systems have also emerged for concurrent programs and for the Ada language in particular. This paper reports on an experience with developing two different approaches, which use symbolic execution, to prove partial correctness and general safety properties of Ada programs. One approach is based upon interleaving the task components while the other is based upon verifying the tasks in isolation and then performing cooperation proofs. Both approaches extend past efforts by incorporating tasking proof rules into the symbolic executor allowing Ada programs with tasking to be formally verified.*

The limitations of each approach are presented, along with each approach's advantages and disadvantages. In particular, the difficulty of dealing with communication statements in a loop structure are addressed in detail.

Dillon88b
Dillon, L.K. "Symbolic Execution-Based Verification of Ada Tasking Programs." *Proceedings of Third International IEEE Conference on Ada Applications and Environments.* Washington, DC: IEEE Computer Society, May 1988, 3-13.

Abstract: *Symbolic execution has been used successfully with sequential programs for generating the* verification conditions *required for correctness proofs. This paper shows how the symbolic execution model for sequential programs can be extended to a tasking subset of Ada. The criteria for correct operation of a concurrent program include* safety properties, *such as mutual exclusion and freedom from deadlock. The extended model, therefore provides a basis for the automatic generation of verification conditions for proving general safety properties of Ada tasking programs.*

Dillon92
Dillon, L.K. *A Visual Execution Model for Ada Tasking.* Technical Report, Department of Computer Science, University of California, Santa Barbara, CA, 1992.

Abstract: *A visual execution model for Ada tasking can help programmers attain a deeper understanding of the tasking semantics. It can illustrate subtelties in semantic definitions that are not apparent in natural language descriptions of Ada tasking, as well as the consequences of choices made in the language design.*

We describe a contour model of Ada tasking that pictorially depicts asynchronous tasks (threads of execution), relationships between the environments in which tasks execute and the manner in which tasks interact. The use of this high-level execution model makes it possible to 'see' what happens during execution of a program. For example, our contour model can illustrate race conditions that arise during execution of programs, the effects of the definitions of task dependence and termination in Ada on inter-task communication and synchronization, and the interplay between these definitions and basic run-time storage management concerns.

The paper provides a high-level introduction to the contour model of Ada tasking and demonstrates its use.

Eckmann85
Eckmann. "INATEST: an Interactive Environment for Testing Formal Specifications." *ACM Software Eng. Notes 10*, 4 (April 1985).

(From the Introduction) Because the cost of formally verifying large software systems is high in both dollars and time, it is often the practice to formally verify only critical requirements. For example, a system may be formally verified to be consistent with a particular security model. However, in addition to these formally verified critical requirements, most systems also have less critical functional requirements that must be satisfied....

During the past twelve months, the Reliable Soft-

ware Group at UCSB has concentrated its effort on the design and implementation of a symbolic execution tool called Inatest.... Inatest is an interactive tool for testing specifications early in the software lifecycle to determine whether the functional requirements for the system being designed can be met. It provides an environment with various modes of operation to be used in testing formal specifications written in Ina Jo....

Eckmann89

Eckmann, S.T. *Ina Flo User Guide.* Unisys Corporation, Culver City, CA, May 1989.

(From the Introduction) Ina Flo is an information flow analysis tool built into the Ina Jo specification language processor. Ina Flo partially automates covert channel analysis of Ina Jo specifications. *Covert channel analysis* is any method for finding, and evaluating the consequences of covert channels in a system.

Eggert89

Eggert, P., Cooper, D. Eckmann, S., J. Gingerich, S. Holtsberg, N. Kelem, and R. Martin. *FDM User Guide.* TM-8486/000/03, Unisys Corporation, Culver City, CA, Sept. 1889.

(From Preface) This guide is for the users of the Unisys Formal Development Methodology (FDM). New users should read the entire guide for instructions about how to use FDM. Experienced users can skip to the last part of Section 4, which describes the changes in the latest FDM version.

Estrin86

Estrin, G., R.S Fenchel, R.R. Razouk, and M.K. Vernon. "SARA (System ARchitect's Apprentice): Modeling, Analysis, and Simulation Support for Design of Concurrent Systems." *IEEE Trans. Software Eng. SE-12,* 2 (1986), 293-311.

Abstract: *An environment to support designers in the modeling, analysis, and simulation of concurrent systems is described. It is shown how a fully nested structure model supports multilevel design and focuses attention on the interfaces between the modules which serve to encapsulate behavior. Using simple examples, the paper indicates how a formal graph model can be used to model behavior in three domains: control flow, data flow, and interpretation. The effectiveness of the explicit environment model in SARA is discussed and the capability to analyze correctness and evaluate performance of a system model are demonstrated. A description of the integral help designed into SARA shows how the designer can be offered consistent use of any new tool introduced to support the design process.*

Feather87

Feather, M.S. "Language Support for the Specification and Development of Composite Systems." *ACM Trans. Prog. Lang. and Syst. 9,* 2 (April 1987), 198-234.

Abstract: *When a complex system is to be realized as a combination of interacting components, devlopement of those components should commence from a specification of the behavior required of the composite systems. A separate specification should be used to describe the decomposition of that system into components. The first phase of implementation from a specification in this style is the derivation of the individual component behaviors implied by these specifications.*

The virtues of this apporach to specification are expounded, and specification language features that are supportive of it are presented. It is shown how these are incorporated in the specification language Gist, which our group has developed. These issues are illustrated in a development of a controller for elevators serving passengers in a multistory building.

Feiertag79

Feiertag, R. and P.G. Neumann. "The Foundations of a Provable Secure Operating System (PSOS)." *National Computer Conference.* Montvale, NJ: AFIPS, 1979, 329-334.

(From the Introduction) PSOS has been designed according to a set of formal techniques embodying the SRI Hierarchical Development Methodology (HDM). HDM has been described elsewhere, ... and thus is only summarized here. The influence of HDM on the security of PSOS is also discussed elsewhere.... In addition, Linden ... gives a general discussion of the impact of structured design techniques on the security of operating systems (including capability systems).

Feiertag80

Feiertag, R.J. *A Technique for Proving Specifications are Multilevel Secure.* CSL-109, Computer Science Laboratory, SRI International, Menlo Park, CA, Jan. 1980.

(From the Introduction) The following sections describe a technique for verifying that a design for an operating system expressed in terms of a formal specification is consistent with a particular model of multilevel security. The technique to be described is mathematically rigorous and, if applied properly, gives assurance that the given design is multilevel secure by this particular model. The technique is supported by a collection of automated tools.

Feldman90
Feldman, M.B. *Language and System Support for Concurrent Programming.* Curriculum Module SEI-CM-25, DTIC: ADA 223760, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa., April 1990.

(From Capsule Description) This curriculum module is concerned with support for concurrent programming provided to the application programmer by operating systems and programming languages. This includes system calls and language constructs for process creation, termination, synchronization, and communication, as well as nondeterministic language constructs such as the selective wait and timed call. Several readily available languages are discussed and compared; concurrent programming using system services of the UNIX operating system is introduced for the sake of comparison and contrast.

Francez80
Francez, N., D.J. Lehmann, and A. Pnueli. "A Linear History Semantics for Distributed Languages." *Twenty-First Annual Symposium on Foundations of Computer Science.* Long Beach, CA: IEEE Computer Society, 1980.

Abstract: *A denotational semantics is given for a distributed language based on communication (CSP). The semantics uses linear sequences of communications to record computations; for any well formed program segment the semantics is a relation between attainable states and the communication sequences needed to attain these states. In binding two or more processes we match and merge the communication sequences assumed by each process to obtain a sequence and state of the combined process. The approach taken here is distinguished by relatively simple semantic domains and ordering.*

Gerhart79
Gerhart, S.L. and D.S. Wile. "Preliminary Report on the Delta Experiment." *Specifications of Reliable Software.* Washington, DC: IEEE Computer Society, April 1979, 198-211.

Abstract: *The ISI Delta Experiment is an effort to specify and verify a piece of real software of moderate complexity and size (roughly 1000 lines). This preliminary report describes: (1) the Delta function, managing the editing of a single file by several users within an operational message processing system; (2) the formal specification of the Delta function in prose and in algebraic axioms; (3) the verification methodology in levels of (a) prose for the system interface level, (b) algebraic axioms for abstract data types, (c) recursive functions for major operations, (d) implementation of the recursive functions in PASCAL with the abstract data types, and (e) implementation of the PASCAL programs in BLISS; and (4) the experience gained in this experiment, both in specification and verification.*

Ghezzi90
Chezzi, C., D. Mandrioli, and A. Morzenti. "TRIO: A Logic Language for Executable Specifications of Real-Time Systems." *Journal of Systems and Software 12,* 2 (Feb. 1990), 107-124.

Abstract: *We motivate the need for a formal specification language for real-time applications and for a support environment providing tools for reasoning about formal specifications. Then we introduced TRIO, a logic-based specification language. TRIO is first introduced informally through examples. Then a formal declarative semantics is provided, which can accommodate a variety of underlying time structures. Finally, the problem of executing TRIO formal specifications is discussed, and a solution is preseted.*

Ghezzi91
Ghezzi, C. and R.A. Kemmerer. "ASTRAL: an Assertion Language for Specifying Realtime Systems." In *Proceedings of the Third European Software Engineering Conference, ESEC '91.* Lecture Notes in Computer Science, no. 550. Berlin. Springer-Verlag, 1991.

Abstract: *ASTRAL is a formal specification language for realtime systems. This paper discusses the rationale of ASTRAL's design and shows how the language builds on previous language experiments. ASTRAL is intended to support formal software development; therefore, the language itself has been formally defined. ASTRAL's specification style is illustrated by discussing a case study taken from telephony.*

Gold79
Gold, B., R. Linde, R. Peeler, M. Schaefer, J. Scheid, and P. Ward. "A Security Retrofit of VM/370." *National Computer Conference.* Montvale, NJ: AFIPS, 1979, 335-344.

(From the Introduction) The VM/370 Security Retrofit Program is a continuing research and development initiative, funded by the Defense Advanced Research Projects Agency (DARPA), with additional funding provided by Canadian Department of National Defense. The program's primary goal is the security retrofit of a popular commercial operating system, VM/370. Two approaches were originally planned: (1) the design of a feasible, formally verified security kernel to VM/370 and (2) a

"hardening" effort to repair known VM/370 penetration weaknesses. It was subsequently decided not to proceed with the VM/370 hardening task because of the uncertainty of the end result: correction of known security flaws does not guarantee the absence of exploitable, but not yet detected, security flaws in the hardened system.

Goldschlag90a

Goldschlag, D.M. "Mechanizing Unity." In *Programming Concepts and Methods*, M. Broy and C.B. Jones, eds. Amsterdam: North-Holland, 1990, 387-414.

Abstract: *This report describes a mechanically verified proof system for concurrent programs. This proof system may be used to mechanically verify the correctness proofs of concurrent programs. Mechanical verification increases the trustworthiness of a prooof.*

This proof system is based on Unity ..., but is defined with respect to an operational semantics of the transition system model of concurrency .... All proof rules are justified by this opeartional semantics. This methodology makes a clear distinction between the theorems in the proof system and the logical inference rules and syntax that define the underlying logic. Since this proof system essentially encodes Unity in a conservative extension of another sound logic, this encoding proves the soundness of Unity.

The proof system has been implemented on the Boyer-Moore prover, a computer program mechanizing the Boyer-Moor logic,... and has been used to mechanically verify the correctness of an n-processor program satisfying both mutual exclusion and absence of starvation. This paper also describes this program and its correctness theorems and presents the key lemmas that aided the mechanical verification. This proof closely resembles a Unity hand proof, but is longer, since all concepts are defined from first principles. This proof system is suiable for the mechanical verification of a wide class of thoerms, since the underlying prover, though automatic, is guided by the user.

Goldschlag90b

Goldschlag, D.M. "Mechanically Verifying Concurrent Programs with the Boyer-Moore Prover." *IEEE Trans. Software Eng. 16*, 3 (Sept. 1990), 1005-1023.

Abstract: *This paper describes a proof system suitable for the mechanical verification of concurrent programs. Mechanical verification, which uses a computer program to validate a formal proof, increases one's confidence in the correctness of the validated proof.*

This proof system is based on Unity ..., and may be used to specify and verify both safety and liveness properties. However, it is defined with respect to an operational semantics of the transition system model of concurrency. Proof rules are simply theorems of this operational semantics. This methodology makes a clear distinction between the theorems in the proof system and the logical inference rules and syntax which define the underlying logic. Since this proof system essentially encodes Unity in another sound logic, and this encoding has been mechanically verified, this encoding proves the soundness of this formalization of Unity.

This proof system has been mechanically verified by the Boyer-Moore prover, a computer program mechanizing the Boyer-Moor logic .... This proof system has been used to mechanically verify the correctness of a distributed algorithm that computes the minimum node value in a tree. This paper also describes this algorithm and its correctness theorems, and presents the key lemmas that aided the mechanical verification. The mechanized proof closely resembles a hand proof, but is longer, since all concepts are defined from first principles. This proof system is suitable for the mechanical verification of a wide class of programs, since the underlying prover, though automatic, is guided by the user.

Good78

Good, D.I., R.M. Cohen, C.G. Hoch, L.W. Hunter, and D.F. Hare. *Report on the Language Gypsy, Version 2.0.* ICSCA-CMP-10, Institute for Computing Science, University of Texas, Austin, TX, Sept. 1978.

Good82

Good, D.I., A.E. Siebert, and L.M. Smith. *Message Flow Modulator Final Report.* Report No. 34, Institute for Computing Science, University of Texas, Austin, TX, Dec. 1982.

Abstract: *The message flow modulator is a formally specified and proved filter program that is applied continuously to a stream of messages flowing from one computer system to another. Messages that pass the filter are passed to their destination. Messages that do not are logged on an audit trail. The modulator has been designed specifically to monitor the flow of security sensitive message traffic from the Ocean Surveillance Information System of the United States Naval Electronic Systems Command.*

The modulator has been designed, specified, and implemented in the Gypsy language. All of the modules, from the highest level of design to the lowest level of coding, has been formally specified and mechanically proved with the Gypsy Verification Environment. The modulator is specifically de-

signed and intended for use in actual filed operation. It has been tested in a simulated operational environment at the Patuxent River Naval Air Test Center with scenarios developed by an independent, external group. With any modification, the proved modulator passed all of these tests on the first attempt.

Good84a

Good, D.I. *Revised Report on Gypsy 2.1.* Institute for Computing Science, University of Texas, Austin, TX, March 1984.

Abstract: *Gypsy is a language for specifying, implementing, and proving computer programs. This document is the revised report on Gypsy 2.1. Gypsy 2.1 includes almost all of Gypsy 2.0 with some extensions and minor modifications.*

Good84b

Good, D.I., B.L. DiVito, and M.K. Smith. *Using the Gypsy Methodology.* Draft Technical Report, Institute for Computing Science, University of Texas, Austin, TX, June 1984.

Abstract: *This report describes how to use the Gypsy methodology for designing and building formally verified systems. The emphasis is on technique and examples.*

Guttag78

Guttag, J.V., E. Horowitz, and D.R. Musser. "Abstract Data Types and Software Validation." *Comm. ACM 21*, 12 (Dec. 1978), 1048-1064.

Abstract: *A data abstraction can be naturally specified using algebraic axioms. The virtue of these axioms is that they permit a representation-independent formal specification of a data type. An example is given which shows how to employ algebraic axioms at successive levels of implementation. The major thrust of the paper is twofold. First, it is shown how the use of algebraic axiomatizations can simplify the process of proving the correctness of an implementation of an abstract data type. Second, semi-automatic tools are described which can be used both to automate such proofs of correctness and to derive an immediate implementation from the axioms. This implementation allows for limited testing of programs at design time, before a conventional implementation is accomplished.*

Hailpern82

Hailpern, B. *Verifying Concurrent Processes Using Temporal Logic.* Lecture Notes in Computer Science, no. 129. Berlin: Springer-Verlag, 1982.

(From the Introduction) In this thesis we present a technique for proving both safety and liveness properties of parallel programs. Safety properties are assertions that must be satisfied by the system state at all times; they are analogous to partial correctness. Liveness properties refer to events that will occur in the future, such as program termination or the eventual execution of an instruction. We describe new tools for verifying programs and heuristics for developing proofs. We demonstrate the applicability of the technique by proving the correctness of a number of algorithms from the literature in the areas of network protocols and resource allocation. The spirit of this thesis, however, is concerned with the design of programs.

Harel88a

Harel, D. "On Visual Formalisms." *Comm. ACM 31*, 5 (May 1988), 514-531.

Abstract: *The higraph, a general kind of diagramming object, forms a visual formalism of topological nature. Higraphs are suited for a wide array of applications to databases, knowledge representation, and most notably, the behavioral specification of complex concurrent systems using the higraph-based language of statecharts.*

Harel88b

Harel, D., H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, and A. Shtul-Trauring. "STATEMATE: A Working Environment for the Development of Complex Reactive Systems." *10th International Conference on Software Engineering.* Washington, DC: IEEE Computer Society, 1988, 396-406.

Abstract: *This paper provides a brief overview of the STATEMATE system, constructed over the past three years by i-Logix Inc., and Ad Cad Ltd. STATEMATE is a graphical working environment, intended for the specification, analysis, design, and documentation of large and complex reactive systems, such as real-time embedded systems, control and communication systems, and interactive software. It enables a user to prepare, analyze and debug diagrammatic, yet precise, descriptions of the system under development from three inter-related points of view, capturing,* structure, functionality *and* behavior. *These views are represented by three graphical languages, the most intricate of which is the language of* statecharts *used to depict reactive behavior over time. In addition to the use of statecharts, the main novelty of STATEMATE is in the fact that it 'understands' the entire descriptions [sic] perfectly, to the point of being able to analyze them for crucial dynamic properties, to carry out rigorous animated executions and simulations of the described system, and to create running code automatically. These features are invaluable when it*

comes to the quality and reliability of the final outcome.

Harrison88

Harrison L.J. and R.A. Kemmerer. "An Interleaving Symbolic Execution Approach For the Formal Verification of ADA Programs with Tasking." *Proceedings of Third International IEEE Conference on Ada Applications and Environments.* Washington, DC: IEEE Computer Society, May 1988, 15-26.

Abstract: *There have been several efforts to use symbolic execution to test and analyze concurrent programs. Recently proof systems have also emerged for concurrent programs and for the Ada language in particular. This paper focuses on using symbolic execution to prove partial correctness and general safety properties of Ada programs. It expands upon last efforts by incorporating tasking proof rules into the symbolic executor allowing Ada programs with tasking to be formally verified.*

Hayes87

Hayes, I., ed. *Specification Case Studies.* Englewood Cliffs, NJ: Prentice-Hall, 1987.

This book is a collected set of case studies based on the use of Z, providing a well-structured introduction to the use of formal methods. The section on specification of the UNIX filing system may involve sufficiently familiar material to provide a good introduction for many students. It is suitable for use by both instructors and students.

Hennessy88

Hennessy, M. *Algebraic Theory of Processes.* Cambridge, MA: MIT Press, 1988.

This book starts with a tutorial about a language very similar to CCS.

Hinke83

Hinke, T., J. Althouse, and R.A. Kemmerer. "SDC Secure Release Terminal Project." *Proceedings of the 1983 Symposium on Security and Privacy.* Washington, DC: IEEE Computer Society, April 1983.

Abstract: *The SDC Secure Release Terminal (SRT) project provides a useful view of the process involved in constructing software whose code is intended to be formally verified to satisfy desired security properties. The purpose of the SRT is to move appropriately classified data from a processing environment at one security level to a processing environment at another level in machine readable form.*

This paper discusses the design process for the SRT which was carried out using the SDC Formal Development Methodology (FDM). The SRT project is the first application of the FDM code level verification capabilities. However, since the code level verification has not yet been performed this paper concentrates on the design problems inherent in targeting a system for code level verification.

Hoare69

Hoare, C.A.R. "An Axiomatic Basis for Computer Programming." *Comm. ACM 12,* 10 (Oct. 1969), 576-580, 583.

Abstract: *In this paper an attempt is made to explore the logical foundations of computer programming by use of techniques which were first applied in the study of geometry and have later been extended to other branches of mathematics. This involves the elucidation of sets of axioms and rules of inference which can be used in proofs of the properties of computer programs. Examples are given of such axioms and rules, and a formal proof of a simple theorem is displayed. Finally, it is argued that important advantages, both theoretical and practical, may follow from a pursuance of these topics.*

This is the original paper on Hoare's method. The important theoretical and practical advantages alluded to in the abstract have indeed followed from a pursuance of the topics of this paper.

Hoare78

Hoare, C. "Communicating Sequential Processes." *Comm. ACM 21,* 8 (Aug. 1978), 666-677.

Abstract: *This paper suggests that input and output are basic primitives of programming and that parallel composition of communicating sequential processes is a fundamental program structuring method. When combined with a development of Dijkstra's guarded command, these concepts are surprisingly versatile. Their use is illustrated by sample solutions of a variety of familiar programming exercises.*

Hoare85

Hoare, C.A.R. *Communicating Sequential Processes.* Englewood Cliffs, NJ: Prentice-Hall, 1984.

This book is the definitive book explaining language, semantics, and the use of CSP.

Holtsberg89

Holtsberg, S., P. Montgomery, and J. Gingerich. *FDM Error Message Reference.* TM-8494/001/01, Unisys Corporation, Culver City, CA, June 1989.

(From Preface) This reference is for users of the Unisys Formal Development Methodology (FDM).... The purpose of this document is to provide a reference for the error messages generated by Release 12.4 of the FDM tools.

Hopcroft79
Hopcroft, J.E. and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Reading, MA: Addison-Wesley, 1979.

This book is the key book used to teach formal languages and automata theory.

ISO89
International Standards Organization. *Information processing systems — Open Systems Interconnection — LOTOS — A formal description technique based on the temporal ordering of observational behavior.* International Standard ISO 8807. Switzerland: International Standards Organization, 1989.

This is the defining document for the international standard LOTOS specification language

Karam91
Karam, G.M. and R.J.A. Buhr. "Temporal Logic-Based Deadlock Analysis for Ada." *IEEE Trans. Software Eng. SE-17,* 10 (Oct. 1989), 1109-1125.

Abstract: *This paper describes an [sic] temporal logic-based specification language and deadlock analyzer for Ada. The deadlock analyzer (along with other analyzers) are intended for use within TimeBench, a concurrent system-design environment with support for Ada. The specification language, COL, uses linear time temporal logic to provide a formal basis for axiomatic reasoning. The deadlock analysis tool uses the reasoning power of COL to demonstrate that Ada designs specified in COL are system-wide deadlock-free; in essence, it uses a specialized theorem prover to deduce the absence of deadlock. The deadlock algorithm is shown to be decidable for finite systems and acceptable otherwise: it is also shown to have a worst-case computational complexity that is exponential with the number of tasks. The analyzer has been implemented in Prolog. Numerous examples are evaluated using the analyzer—the examples vary in complexity and in the number of tasks: readers and writers, gas station, five dining philosphers, and a layered communications system. The results indicate that analysis time is reasonable for moderate designs in spite of the worst-case complexity of the algorithm.*

Kemmerer82
Kemmerer, R.A. *Formal Verification of an Operating System Security Kernel.* Ann Arbor, MI: UMI Research Press, 1982.

Kemmerer83
Kemmerer, R.A. and S.T. Eckmann. *A User's Manual for the UNISEX System.* TRCS83-05, Department of Computer Science, University of California, Santa Barbara, CA, Dec. 1983.

(From the Introduction) UNISEX, a UNIx-based Symbolic EXecutor for Pascal, provides an environment for testing and verification of programs.

Kemmerer85
Kemmerer, R.A. and Eckmann, S.T. "UNISEX: A UNIx-based Symbolic EXecutor for Pascal." *Software—Practice and Experience 15,* 5 (May 1985), 439-458.

Abstract: *UNISEX is a UNIX-based symbolic executor for Pascal. The UNISEX system provides an environment for both testing and verifying Pascal programs. The system supports a large subset of Pascal, runs on UNIX and provides the user with a variety of debugging features to help in the difficult task of program validation. This paper contains a brief introduction to symbolic execution, followed by an overview if the features of UNISEX, a discussion of the UNISEX Pascal language, and some of the implementation details for the UNISEX system. Finally, some of the problems encountered when designing and implementing the system are discussed as well as future directions.*

King80
King, J.C. "Program Correctness: On Inductive Assertion Methods." *IEEE Trans. Software Eng. SE-6* (1980), 465-479.

Abstract: *A study of several of the proof of correctness methods is presented. In particular, the form of induction used is explored in detail. A relational semantic model for programming languages is introduced and its relation to predicate transformers is explored. A rather elementary viewpoint is taken in order to expose, as simply as possible, the basic differences of the methods and the underlying principles involved. These results were obtained by attempting to thoroughly understand the "subgoal induction" method.*

Klein83
Klein, M. *Department of Defense Trusted Computer System Evaluation Criteria.* Fort Meade, MD: Department of Defense, 1983.

(From Preface) The trusted computer system evaluation criteria defined in this document classify systems into four broad hierarchical divisions of enhanced security protection. They provide a basis for the evaluation of effectiveness of security controls built into automatic data processing system

products. The criteria were developed with three objectives in mind: (a) to provide users with a yardstick with which to assess the degree of trust that can be placed in computer systems for the secure processing of classified or other sensitive information; (b) to provide guidance to manufacturers as to what to build into their new, widely-available trusted commercial products in order to satisfy trust requirements for sensitive applications; and (c) to provide a basis for specifying security requirements in acquisition specifications. Two types of requirements are delineated for secure processing: (a) specific security feature requirements and (b) assurance requirements.

Lamport76

Lamport, L. "The Synchronization of Independent Processes." *Acta Informatica 6* (1976), 15-34.

Abstract: *This paper considers the problems of programming a multiple process system so that it continues to operate despite the failure of individual processes. A powerful synchronizing primitive is defined, and it is used to solve some sample problems. An algorithm is then given which implements this primitive under very weak assumptions about the nature of interprocess communication, and a careful informal proof of its correctness is given.*

Lamport80a

Lamport, L. "The 'Hoare Logic' of Concurrent Programs." *Acta Informatica 14* (1980), 21-37.

Abstract: *Hoare's logical system for specifying and proving partial correctness properties of sequential programs is generalized to concurrent programs. The basic idea is to define the assertion $\{P\}S\{Q\}$ to mean that if execution is begun anywhere in S with P true, then P will remain true until S terminates, and Q will remain true if and when S terminates. The predicates P and Q may depend upon program control locations as well as upon the values of variables. A system of inference rules and axiom schemas is given, and a formal correctness proof for a simple program is outlined. We show that by specifying certain requirements for the unimplemented parts, correctness properties can be proved without completely implementing the program. The relation to Pnueli's temporal logic formalism is also discussed.*

Lamport80b

Lamport, L. "'Sometime' Is Sometimes 'Not Never', On the Temporal Logic of Programs." *Conference Record Seventh Annual ACM Symposium on Principles of Programming Languages.* New York: ACM, Jan. 1980, 174-184.

(From the Introduction) We will describe two different temporal logics for reasoning about a computational model. The same formulas appear in both logics, but they are interpreted differently. The two interpretations correspond to two ways of viewing time: as a continually branching set of possibilities, or as a single linear sequence of actual events. The temporal concepts of "sometime" and "not never" ("not always not") are equivalent in the theory of linear time, but not in the theory of branching time — hence, our title. We will argue that the logic of linear time is better for reasoning about concurrent programs, and the logic of branching time is better for reasoning about nondeterministic programs.

Lamport83a

Lamport, L. "Specifying Concurrent Program Modules." *ACM Trans. Prog. Lang. and Syst. 5*, 2 (Feb. 1983), 190-222.

Abstract: *A method for specifying program modules in a concurrent program is described. It is based upon temporal logic, but it uses new kinds of temporal assertions to make the specifications simpler and easier to understand. The semantics of the specifications is described informally, and a sequence of examples are given culminating in a specification of three modules comprising the alternating-bit communication protocol. A formal semantics is given in the appendix.*

Lamport83b

Lamport, L. "What Good is Temporal Logic?" In *Information Processing 83*, R.E.A. Mason, ed. Amsterdam: North-Holland, 1983, 657-667.

Abstract: *Temporal Logic is a formal system for specifying and reasoning about concurrent programs. It provides a uniform framework for describing a system at any level of abstraction, thereby supporting hierarchical specification and verification.*

Lamport84

Lamport, L. "What it Means for a Concurrent Program to Satisfy a Specification: Why No One Has Specified Priority." *Conference Record Twelfth Annual ACM Symposium on Principles of Programming Languages.* New York: ACM, Jan. 1985, 78-83.

Abstract: *The formal correspondence between an implementation and its specification is examined. It is shown that existing specifications that claim to describe priority are either vacuous or else too restrictive to be implemented in some reasonable situations. This is illustrated with a precisely formulated problem of specifying a first-come-first-served mutual exclusion algorithm, which it is claimed cannot be solved by existing methods.*

Lamport86
Lamport, L. *A Simple Approach To Specifying Concurrent Systems.* Technical Report 15, Digital Systems Research Center, Palo Alto, CA, 1986.

Abstract: In the transition axiom method, safety properties of a concurrent system can be specified by programs: liveness properties are specified by assertions in a simple temporal logic. The method is described with some simple examples, and its logical foundation is informally explored through a careful examination of what it means to implement a specification. Language issues and other practical details are largely ignored.

Lamport87
Lamport, L. *win and sin: Predicate Transformers for Concurrency.* Technical Report 17, Digital Systems Research Center, Palo Alto, CA, May 1987.

Abstract: Dijkstra's weakest liberal precondition *and* strongest postcondition *predicate transformers are generalized to the* weakest invariant *and* strongest invariant. *These new predicate transformers are useful for reasoning about concurrent programs containing operations in which the grain of atomicity is unspecified. they can also be used to replace behavioral arguments with more rigorous assertional ones.*

Landwehr81
Landwehr, C.E. "Formal Models for Computer Security." *ACM Computing Surveys 13,* 3 (Sept. 1981), 247-278.

Abstract: Efforts to build "secure" computer systems have now been underway for more than a decade. Many designs have been proposed, some prototypes have been constructed, and a few systems are approaching the production stage. A small number of systems are even operating in what the Department of Defense calls "multilevel" mode: some information contained in these computer systems may even have a classification higher than the clearance of some of the users of those systems.

This paper reviews the need for formal security models, describes the structure and operation of military security controls, considers how automation has affected security problems, surveys models that have been proposed and applied to date, and suggests possible directions for future models.

Landwehr83
Landwehr, C. "The Best Available Technologies for Computer Security." *Computer 16,* 7 (July 1983).

Abstract: This concise overview of secure system developments summarizes past and current projects, deciphers computer security lingo, and offers advice to prospective designers.

Lee81
S. Lee and S.L. Gerhart. *AFFIRM User's Guide.* USC Information Sciences Institute, Marina del Rey, CA, Feb. 1981.

(From Preface) The *AFFIRM* USER'S GUIDE accompanies the *AFFIRM* REFERENCE MANUAL and the *AFFIRM* TYPE LIBRARY in order to make life easier for people who really want to use *AFFIRM.* The GUIDE is a distillation of experience by the PV project (and others) which we want to pass on to users.

Leveson83
Leveson, N.G., A.I Wasserman, and D.M. Berry. "BASIS: A Behavioral Approach to the Specification of Information Systems." *Information Sciences 8,* 1 (1983), 15-23.

Abstract: This paper is an overview of BASIS (Behavioral Approach to the Specification of Information Systems), a multi-step formal method used for information systems design and development. The steps include information analysis, semantic specification, verification of the specification, concrete implementation, and verification of the implementation. In this way, BASIS can be used to provide a formal basis for information systems development. We provide an example showing how BASIS can be used in conjunction with implementation in the programming language PLAIN.

Leveson91
Leveson, N.G., M. Heimdahl, H. Hildreth, J. Reese, and R. Ortega. "Experiences using Statecharts for a System Requirements Specification." *Sixth International Workshop on Software Specification and Design.* Washington, DC: IEEE Computer Society, 1991, 31-41.

Abstract: This paper describes some lessons learned and issues raised while building a system requirements specification for a real aircraft collision avoidance system using statecharts. Some enhancements to statecharts were necessary to model the complete system and a few notational changes were made to improve reviewability.

Levitt85
Levitt, K.N. *Communications Network in Revised SPECIAL.* Computer Science Laboratory, SRI International, Menlo Park, CA, June 1985.

Lindsay88
Lindsay, P.A. "A Survey of Mechanical Support for Formal Reasoning." *Software Engineering Journal 3* (1988), 3-27.

This survey examines seven support systems in detail and introduces eleven others. The systems discussed in detail are LCF (Logic for Computable Functions), NuPRL, Veritas, Isabelle, AFFIRM, the Boyer-Moore system, and Gypsy. The bibliography contains 87 items.

Liskov74

Liskov, B.H. and S.N. Zilles. "Programming with Abstract Data Types." *ACM SIGPLAN Notices 9*, 4 (April 1974), 50-60.

Abstract: *The motivation behind this work in very-high-level languages is to ease the programming task by providing the programmer with a language containing primitives or abstractions suitable to his problem area. The programmer is then able to spend his effort in the right place; he concentrates on solving his problem, and the resulting program will be more reliable as a result. Clearly, this is a worthwhile goal.*

Unfortunately, it is very difficult for a designer to select in advance all the abstractions which the users of his language might need. If a language is to be used at all, it is likely to be used to solve problems which its designers did not envision, and for which the abstractions embedded in the language are not sufficient.

This paper presents an approach which allows the set of built-in abstractions to be augmented when the need for a new data abstraction is discovered. This approach to the handling of abstractions is an outgrowth of work on designing a language for structured programming. Relevant aspects of this language are described, and examples of the use and definitions of abstractions are given.

Locasso80

Locasso, R., J. Scheid, D.V. Schorre, and P.R. Eggert. *The Ina Jo Reference Manual.* TM-(L)-6021/001/000, System Development Corporation, Santa Monica, CA, June 1980.

Abstract: *The Ina Jo specification language is in use at SDC as part of its formal development method. System specifications written in Ina Jo language are verified mechanically with respect to user-defined criteria. An Ina Jo specification is a collection of levels; each level describes an abstract machine by describing its states and possible state transitions. Lower levels contain mappings describing how they are intended to implement parts of higher levels. The top level contains correctness requirements that must be met by the entire system. Thus an Ina Jo specification allows for a structured proof of the desired properties of a complete system.*

Lucas69

Lucas, P. and K. Walk. "On the Formal Description of PL/I." *Annual Review in Automatic Programming 6*, 3 (1969).

(From the Introduction) This paper presents tools and design criteria for the formal description of programming languages. The results reported were achieved mainly during the development of the formal definition of PL/I as documented in a series of Technical Reports [from IBM Vienna Laboratories]. An appropriately tailored subset of PL/I is used to illustrate these results. Their applicability is, however, not restricted to PL/I.

Luckham86

Luckham, D.C., D.P. Helmbold, S. Meldal, D.L. Bryan, and M.A. Haberler. "Task Sequencing Language for Specifying Distributed Ada Systems." In *System Development and Ada: CRAI Workshop on Software Factories and Ada.* Lecture Notes in Computer Science, no. 275. Berlin: Springer-Verlag, 1986.

Abstract: *TSL-1 is a language for specifying sequences of tasking events occuring in the execution of distributed Ada programs. Such specifications are intended primarily for testing and debugging of Ada tasking programs, although they can also be applied in designing programs. TSL-1 specifications are included in an Ada program as formal comments. They express constraints to be satisfied by the sequences of acutal tasking events. An Ada program is consistent with its TSL-1 specifications if its runtime behavior always satisfies them. This paper presents an overview of TSL-1. The features of the language are described informally, and examples illustrating the use of TSL-1, both for debugging and for specification of tasking programs, are given. A definition of robust TSL-1 specifications that takes into account uncertainty in runtime observation of behavior of distributed systems is given. A runtime monitor for checking consistency of an Ada program with TSL-1 specifications has been implemented. In the future, constructs for defining abstract tasks will be added to TSL-1, forming a new languages, TSL-2, for the specification of distributed systems prior to their implementation in any particular programming language.*

Manna72

Manna, Z., S. Ness, and J. Vuillemin. "Inductive Methods for Proving Properties of Programs." *ACM SIGPLAN Notices 7*, 1 (Jan. 1972), 27-50.

Abstract: *We have two main purposes in this paper. First, we clarify and extend known results about computation of recursive programs, emphasizing the difference between the theoretical and*

practical approaches. Secondly, we present and examine various methods for proving properties of recursive programs. We discuss in detail two powerful inductive methods, computational induction and structural induction, illustrating their applications by various examples. We also briefly discuss some other related methods.

Our aim in this work is to introduce inductive methods to as wide a class of readers as possible and to demonstrate their power as practical techniques. We ask the forgiveness of our more theoretical-minded colleagues for our occasional choice of clarity over precision.

Manna74

Manna, Z. *Mathematical Theory of Computation.* New York: McGraw-Hill, 1974.

The book has a wide-ranging survey of topics in the theory of computation. Chapter 3 deals with verification of program correctness and halting.

Manna81

Manna, Z. and A. Pnueli. "Verification of Concurrent Programs: The Temporal Framework." In *The Correctness Problem in Computer Science*, R.S. Boyer and J.S. Moore, eds. London: Academic Press, 1981, 215-273.

Abstract: *This is the first in a series of reports describing the application of Temporal Logic to the specification and verification of concurrent programs.*

We first introduce Temporal Logic as a tool for reasoning about sequences of states. Models of concurrent programs based both on transition graphs and on linear-text representations are presented and the notions of concurrent and fair executions are defined.

The general temporal language is then specialized to reason about those execution states and execution sequences that are fair computations of concurrent programs. Subsequently, the language is used to describe properties of concurrent programs.

The set of interesting properties is classified into Invariance (Safety), Eventuality (Liveness) and Precedence (Until) properties. Among the properties studied are: Partial Correctness, Global Invariance, Clean Behavior, Mutual Exclusion, Deadlock Absence, Termination, Total Correctness, Intermittent Assertions, Accessibility, Starvation Freedom, Responsiveness, Safe Liveness, Absence of Unsolicited Response, Fair Responsiveness and Precedence.

In the following reports of this series we use the temporal formalism to develop proof methodologies for proving the properties discussed here.

McCarthy65

McCarthy, J., P.W. Abrahams, D.J. Edwards, T.P. Hart, and M. Levin. *LISP 1.5 Programmer's Manual.* Cambridge, MA: MIT Press, 1965.

This book is the original LISP manual. It contains a definition of LISP written in LISP.

McCauley79

McCauley, E. and P. Drongowski. "KSOS — The Design of a Secure Operating System." *National Computer Conference.* Montvale, NJ: AFIPS, 1979, 345-353.

(From the Introduction) This paper discusses the design of the Department of Defense (DoD) Kernelized Secure Operating System (KSOS, formerly called Secure UNIX). KSOS is intended to provide a provably secure operating system for larger minicomputers, KSOS will provide a system interface closely compatible with the UNIX operating system. The initial implementation of KSOS will be on a Digital Equipment Corporation PDP-11/70 computer system. A group from Honeywell is also proceeding with an implementation for a modified version of the Honeywell Level 6 computer system.

McGowan71

McGowan, C.L. "An Inductive Proof Technique for Interpreter Correctness." In *Courant Computer Science Symposium on Formal Semantics of Programming Languages*, R. Rustin, ed. Englewood Cliffs, NJ: Prentice-Hall, 1971.

Abstract: *A general inductive proof technique is presented which has been successfully used in establishing the correctness and equivalence of interpreters for the lambda calculus and for block structured languages.*

Meadows88

Meadows, C.A. *A Method for Automatically Translating Trace Specification into Prolog.* NRL9131, Naval Research Laboratory, 1988.

Milner80

Milner, R. *A Calculus of Communicating Systems.* Lecture Notes in Computer Science, no. 92. Berlin: Springer-Verlag, 1980.

This book is the definitive book defining CCS.

Milner89

Milner, R. *Communication and Concurrency.* Englewood Cliffs, NJ: Prentice-Hall, 1989.

This book is the definitive book explaining and defining CCS. It is significantly more tutorial than the other book by the same author in 1980.

Moore90
Moore, A.P. "The Specification and Verified Decomposition of System Requirements Using CSP." *IEEE Trans. Software Eng. 16*, 3 (Sept. 1990), 933-948.

Abstract: *An important principle of building trustworthy systems is to rigorously analyze the critical requirements early in the development process, even before starting system design. Existing proof methods for systems of communicating processes focus on the bottom-up composition of component-level specifications into system-level specifications. Trustworthy system development requires, instead, the top-down derivation of component requirements from the critical system requirements. This paper describes a formal method for decomposing the requirements of a system into requirements of its component processes and a minimal, possibly empty, set of synchronization requirements. The Trace Model of Hoare's Communicating Sequential Processes (CSP) is the basis for the formal method. We apply the method to an abstract voice transmitter and describe the role that the EHDM verification system plays in the transmitter's decomposition. In combination with other verification techniques, we expect that the method defined here will promote the development of more trustworthy systems.*

Morgan87
Morgan, E.T. and R.R. Razouk. "Interactive State-Space Analysis of Concurrent Systems." *IEEE Trans. Software Eng. SE-13*, 10 (Oct. 1987), 1080-1091.

Abstract: *The introduction of concurrency into programs has added to the complexity of the software design process. This is most evident in the design of communications protocols where concurrency is inherent to the behavior of the* system. *The complexity exhibited by such software systems makes more evident the need for computer-aided tools for automatically analyzing behavior.*

The Distributed Systems project at UCI has been developing techniques and tools, based on Petri nets, which support the design and evaluation of concurrent software systems. Techniques based on constructing reachability graphs that represent projections and selections of complete state-spaces have been developed. This paper focuses attention on the computer-aided analysis of these graphs for the purpose of proving correctness of the modeled system. The application of the analysis technique to evaluating simulation results for correctness is discussed. The tool which supports this analysis (the reachability graph analyzer, RGA) is also described. This tool provides mechanisms for proving general system properties (e.g., deadlock-freeness) as well as system-specific properties. The tool is sufficiently general to allow a user to apply complex user-defined analysis algorithms to reachability graphs. The alternating-bit protocol, with a bounded channel, is used to demonstrate the power of the tool and to point to future extensions.

Musser85a
Musser, D.R. and D.A. Cyrluk. *AFFIRM-85 Installation Guide and Reference Manual Update.* General Electric Corporate Research and Development, Schenectady, NY, March 1985.

Musser85b
Musser, D.R. "Aids to Hierarchical Specification Structuring and Reusing Theorems in Affirm-85." *ACM Software Eng. Notes 10*, 4 (1985).

(From the Introduction) The AFFIRM Program Verification System originated at the University of Southern California Information Sciences Institute (ISI). It is an experimental system for the algebraic specification and verification of abstract data types and Pascal-like programs using these types.... AFFIRM-85 is an enhanced version of AFFIRM that is being developed at General Electric Corporate Research and Development Center (GE-CRD). This paper briefly describes the two major extensions that will completed in early in 1985. The primary purpose of these and several minor extensions is to enable the use of AFFIRM in carrying out a larger part of the software development process than previously has been possible.

Neumann77
Neumann, P.G., R.S. Boyer, R.J. Feiertag, and K.N. Levitt. *A Provably Secure Operating System: The System, Its Applications, and Proofs.* Final Report, Computer Science Laboratory, SRI International, Menlo Park, CA, Feb. 1977.

Abstract: *This report provides a detailed description of the design of a secure operating system and some of its applications, along with proofs of some of the properties related to security. Discussed here are:*

- *a formal methodology for the design and implementation of computer operating systems and application subsystems, and for the formal proof of properties of such systems, with respect to both the design and the implementation;*
- *the design of a secure capability-based operating system according to this methodology to meet advanced security requirements, together with relevant implementation considerations;*
- *the design of several application subsystems for this operating system, including support for multilevel security classifica-*

tions, for confined subsystems, for a secure relational data management system, and for monitoring of security;

- *the statement and proof of properties of the design for the operating system and for certain application subsystems;*
- *an evaluation of the significance of this work, and considerations for the future development of secure systems and subsystems.*

Nguyen84

Nguyen, V., D. Gries, and S. Owicki. "A Model for Temporal Proof System for Networks of Process." *Conference Record Twelfth Annual ACM Symposium on Principles of Programming Languages.* New York: ACM, Jan. 1985, 121-131.

Abstract: *A model and a sound and complete proof system for networks of processes in which component processes communicate exclusively through messages is given. The model, an extension of the trace model, can describe both synchronous and asynchronous networks. The proof system uses temporal-logic assertions on sequences of observations—a generalization of traces. The use of observations (traces) makes the proof system simple, compositional and modular, since internal details can be hidden. The expressive power of temporal logic makes it possible to prove temporal properties (safety, liveness, precedence, etc.) in the system. The proof system is language-independent and works for both synchronous and asynchronous networks.*

Nielsen89

Nielsen, M., K. Havelund, K.R. Wagner, and C. George. "The RAISE Language, Method and Tools." *Formal Aspects of Computing 1*, 1 (1989), 85-114.

Abstract: *This paper presents the RAISE software development method, its associated specification language, and the tools supporting it, The RAISE method enables the stepwise development of both sequential and conccurrent software from abstract specification through design to implementation. All stages of RAISE software development are expressed in the wide-spectrum RAISE specification language. The RAISE tools form an integrated tool environment supporting both language and method.*

The paper surveys RAISE and furthermore, more detailed presentations of major RAISE results are provided. The subjects of these are (a) an example of the use of the RAISE method and language, and (b) a presentation of the mathematical semantics of the RAISE specification language.

Organick72

Organick, E.I. *The Multics System.* Cambridge, MA: MIT Press, 1972.

This book examines the structure of the MIT multics system from the bottom upwards.

Organick78

Organick, E.I., A.I. Forsythe, and R.P. Plummer. *Programming Language Structures.* New York, NY: Academic Press, 1978.

This book uses Johnston's Contour Model, a pictorial information structure model to describe a variety of programming languages and their features.

Owicki76a

Owicki, S. and D. Gries. "Verifying Properties of Parallel Programs: An Axiomatic Approach." *Comm. ACM 19*, 5 (May 1976), 279-285.

Abstract: *An axiomatic method for proving a number of properties of parallel programs is presented. Hoare has given a set of axioms for partial correctness, but they are not strong enough in most cases. This paper defines a more powerful deductive system which is in some sense complete for partial correctness. A crucial axiom provides for the use of auxiliary variables, which are added to a parallel program as an aid to proving it correct. The information in a partial correctness proof can be used to prove such properties as mutual exclusion, freedom from deadlock, and program termination. Techniques for verifying these properties are presented and illustrated by application to the dining philosophers problem.*

Owicki76b

Owicki, S. and D. Gries. "An Axiomatic Proof Technique for Parallel Programs I." *Acta Informatica 6* (1976), 319-340.

Abstract: *A language for parallel programming, with a primitive construct for synchronization and mutual exclusion, is presented. Hoare's deductive system for proving partial correctness of sequential programs is extended to include the parallelism described by the language. The proof method lends insight into how one should understand and present parallel programs. Examples are given using several of the standard problems in the literature. Methods for proving termination and the absence of deadlock are also given.*

Owicki82

Owicki, S. and L. Lamport. "Proving Liveness Properties of Concurrent Programs." *ACM Trans. Prog. Lang. and Syst. 4*, 3 (Oct. 1982), 455-495.

Abstract: *A liveness property asserts that program execution eventually reaches some desirable state. While termination has been studied extensively, many other liveness properties are important for concurrent programs. A formal proof method, based on temporal logic, for deriving liveness properties is presented. It allows a rigorous formulation of simple informal arguments. How to reason with temporal logic and how to use safety (invariance) properties in proving liveness is shown. The method is illustrated using, first, a simple programming language without synchronization primitives, then one with semaphores. However, it is applicable to any programming language.*

Paolini81

Paolini, P. *Abstract Data Types and Data Bases.* Ph.D. Th., Computer Science Department, University of California, Los Angeles, CA, 1981.

Parnas72a

Parnas, D.L. "On the Criteria to be Used in Decomposing Systems into Modules." *Comm. ACM 15*, 2 (Dec. 1972), 1053-1058.

Abstract: *This paper discusses modularization as a mechanism for improving the flexibility and comprehensibility of a system while allowing the shortening of its development time. The effectiveness of a "modularization" is dependent on the criteria used in dividing the system into modules. A system design problem is presented and both a conventional and unconventional decomposition are described. It is shown that the unconventional decompositions have distinct advantages for the goals outlined. The criteria used in arriving at the decomposition are discussed. The unconventional decomposition, if implemented with the conventional assumption that a module consists of one or more subroutines, will be less efficient in most cases. An alternative approach to implementation which does not have this effect is sketched.*

Parnas72b

Parnas, D.L. "A Technique for the Specification of Software Modules." *Comm. ACM 15*, 5 (May 1972), 330-336.

Abstract: *This paper presents an approach to writing specifications for parts of software systems. The main goal is to provide specifications sufficiently precise and complete that other pieces of software can be written to interact with the piece specified without additional information. The secondary goal is to include in the specification no more information than necessary to meet the first goal. The technique is illustrated by means of a variety of examples from a tutorial system.*

Pedersen89

Pedersen, J.S. *Software Development Using VDM.* Curriculum Module SEI-CM-16, DTIC: ADA 235996, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa., Dec. 1989.

(From Capsule Description) This module introduces the Vienna Definition Method (VDM) approach to software development. The method is oriented toward a formal model view of the sofware to be developed. The emphasis of the module is on formal specification and systematic development of programs using VDM. A major part of the module deals with the particular specification language (and abstraction mechanisms) used in VDM.

Peterson81

Peterson, J.L. *Petri Net Theory and the Modeling of Systems.* Englewood Cliffs, NJ: Prentice-Hall, 1981.

This is the classic book covering Petri Nets and their use in modeling of concurrent systems.

Place90

Place, P.R.H., W.B. Wood, and M. Tudball. *Survey of Formal Specification Techniques for Reactive Systems.* CMU/SEI-90-TR-5, DTIC: ADA 22374, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1990.

Abstract: *Formal methods are being considered for the description of many systems including systems with real-time constraints and multiple concurrently executing processes. This report develops a set of evaluation criteria and evaluates Communicating Sequential Processes (CSP), the Vienna Definition Method (VDM), and temporal logic. The evaluation is based on specifications, written with each of the techniques, of an example avionics system.*

Plotkin76

Plotkin, G.D. "A Power Domain Construction." *SIAM Journal of Computing 5*, 3 (1976), 452-487.

Abstract: *We develop a powerdomain construction $\mathcal{P}(\bullet)$, which is analagous to the powerset construction and also fits in with the usual sum, product and exponentiation constructions on domains. The desire for such a construction arises when considering programming languages with nondeterministic features or parallel features treated in a nondeterministic way. We hope to achieve a natural, fully abstract semantics in which such equivalences as* (p par q) = (q par p) *hold. The domain* ($D \rightarrow$*Truthvalues*) *is not the right one, and instead we take the (finitely) generable subsets of D. When D is discrete they are ordered in an elementwise fashion. In the general case they are given the*

coarsest ordering consistent, in an appropriate sense, with the ordering given in the discrete case. We then find a restricted class of algebraic inductive partial orders which is closed under $\mathcal{P}(\bullet)$ as well as the sum, product and exponentiation constructions. This class permits the solution of recursive domain equations, and we give some illustrative semantics using $\mathcal{P}(\bullet)$.

It remains to be seen if our powerdomain construction does give rise to fully abstract semantics, although such natural equivalences as the above do hold. The major deficiency is the lack of a convincing treatment of the fair parallel construct.

Plotkin83

Plotkin, G.D. "An Operational Semantics for CSP." In *Formal Description of Programming Concepts II*, D. Bjorner, ed. Amsterdam: North-Holland, 1983, 199-224.

Abstract: *Hoare's CSP is used to illustrate a method employing the well-known idea of* labelled transition systems *to provide operational semantics for programming languages. What is new is their* specifications*; following the modern emphasis on structure they are given by* structural induction *on abstract syntax, resulting in a precise but intuitive semantics. Most of CSP is treated including the arbitrary nesting of parallel commands and the failure convention when communicating with a terminated process; also a solution to the library problem is proposed.*

Pnueli77

Pnueli, A. "The Temporal Logic of Programs." *Eighteenth Annual Symposium on the Foundations of Computer Science.* Long Beach, CA: IEEE Computer Society, Nov. 1977.

Abstract: *A unified approach to program verification is suggested, which applies to both sequential and parallel programs. The main proof method suggested is that of temporal reasoning in which the time dependence of events is the basic concept. Two formal systems are presented for providing a basis for temporal reasoning. One forms a formalization of the method of intermittent assertions, while the other is an adaptation of the tense logic system K_t, and is particularly suitable for reasoning about concurrent programs.*

Pnueli81

Pnueli, A. "The Temporal Semantics of Concurrent Programs." *Theoretical Computer Science 13* (1981), 45-60.

Abstract: *The formalism of Temporal Logic is suggested as an appropriate tool for formalizing the semantics of concurrent programs. A simple model of concurrent programs is presented in which n processors are executing concurrent n disjoint programs under a shared memory environment. The semantics of such a program specifies the class of state sequences which are admissible as proper execution sequences under the program. The two main criteria which are required are:*

- *Each State is obtained from its predecessor in the sequence by exactly one processor performing an atomic instruction in its process.*
- *Fair Scheduling: No processor which is infinitely often enabled will be indefinitely delayed.*

The basic elements of temporal Logic are introduced in a particular logic framework DX. The usefulness of Temporal Logic notation in describing properties of concurrent programs is demonstrated. A construction is then given for assigning to a program P *a temporal formula* W (P) *which is true on all proper execution sequences of P. In order to prove that a program* P *possesses a property* R*, one has only to prove the implications* W (P) $\subset$ R*. An example of such proof is given. It is then demonstrated that specification of the Temporal character of the program's behavior is absolutely essential for the unambiguous understanding of the meaning of programming constructs.*

Popek75

Popek, G.J. and C.S. Kline. "A Verifiable Protection System." *ACM SIGPLAN Notices 10*, 6 (June 1975), 294-304.

Abstract: *This paper reports on the design and implementation of the UCLA Virtual Machine System, a multiuser operating system base that has been developed to provide ultra high reliability protection and security. Details are presented of the UCLA-VM system, a prototype of which now exists. Concepts which have influenced its structure are discussed, including program verification, security kernels, virtual machines, virtual memory, and the need for flexible information sharing facilities. A new mechanism,* capability faulting*, is developed in order to remove much of the virtual memory support from the security kernel. Flexible, reliable control of sharing is obtained by extensions to several of these concepts, especially through the use of* levels of kernels.

Popek79

Popek, G., M. Kampe, C. Kline, A. Stoughton, M. Urban, and E. Walton. "UCLA Secure Unix." *National Computer Conference.* Montvale, NJ: AFIPS, 1979, 355-364.

(From the Introduction) The UCLA Data Secure Unix [sic] operating system is intended as a demon-

stration that verifiable data security with general functionality is attainable today in medium scale computing systems. More specifically, the UCLA system has the characteristic that data security, the assurance that data cannot be directly read or modified without specific permission, is enforced via a limited amount of kernel software. High levels of care are being applied to demonstrate that the security properties of that software are correctly implemented. In addition, the system is designed so that confinement can be demonstrated by audit of some additional, isolated code.

Razouk77

Razouk, R.R. *The GMB Simulator System Reference Manual.* Computer Science Department, University of California, Los Angeles, CA, July 1977.

Razouk79

Razouk, R.R., M. Vernon, and G. Estrin. "Evaluation Methods in SARA — The Graph Model Simulator." *Proceedings of the Conference on Simulation, Measurement and Modeling of Computer Systems.* Washington, DC: IEEE Computer Society, Aug. 1979, 189-206.

Abstract: The supported methodology evolving in the SARA (System ARchitects' Apprentice) system creates a design framework on which increasingly powerful analytical tools are to be grafted. Control flow analyses and program verification tools have shown promise. However, in the realm of the complex systems which interest us there is a great deal of research and development to be done before we can count on the use of such powerful tools. We must always be prepared to resort to experiments for evaluation of proposed designs.

This paper describes a fundamental SARA tool, the graph model simulator. During top-down refinement of a design, the simulator is used to test consistency between the levels of abstraction. During composition, known building blocks are linked together and the composite graph model is tested relative to the lowest top-down model. Design of test environments is integrated with the multilevel design process. The SARA methodology is exemplified through design of a higher level building block to do a simple FFT.

Razouk80

Razouk, R.R., M. Vernon, and M. Brewer. *Control-Flow Analyzer Reference Manual.* Computer Science Department, University of California, Los Angeles, CA, Feb. 1980.

Razouk85a

Razouk, R.R. and C.V. Phelps. "Performance Analysis Using Timed Petri Nets." In *Protocol Specifications, Testing, and Verification, IV,* Y. Yemini, R. Strom, and S. Yemini, eds. Amsterdam: North-Holland, 1985.

Abstract: Petri Nets have been successfully used to model and evaluate the performance of distributed systems. Several researchers have extended the basic Petri Net model to include time, and have demonstrated that restricted classes of Petri Nets can be analyzed efficiently. Unfortunately, the restrictions prohibit the techniques from being applied to many interesting systems, e.g., communication protocols. This paper proposes a version of timed Petri Nets which accurately models communication protocols, and which can be analyzed using Timed Reachability Graphs. Procedures for constructing and analyzing these graphs are presented. The analysis is shown to be applicable to a larger class of Timed Petri Nets than previously thought. The model and the analysis technique are demonstrated using a simple communication protocol.

Razouk85b

Razouk, R.R. and D.S. Hirschberg. "Tools for Efficient Analysis of Concurrent Software Systems." *Proceedings of SOFTFAIR 85 Conference on Software Development Tools, Techniques and Alternatives,* Washington, DC: IEEE Computer Society, Dec. 1985, 192-198.

Abstract: The ever increasing use of distributed computing as a method of providing added computing power and reliability has sparked interest in methods to model and analyze concurrent hardware/software systems. Efficient automated analysis tools are needed to aid designers of such systems. The Distributed Systems Project at UCI has been developing a suite of tools (dubbed the P-NUT system) which supports efficient analysis of concurrent software. This paper presents the principles which guide the development of P-NUT tools and discusses the development of one of the tools: the Reachability Graph Builder (RGB). The P-NUT approach to tool development has resulted in the production of a highly efficient tool for constructing reachability graphs. The careful design of data structures and associated algorithms has significantly enlarged the class of models which can be analyzed.

Reynolds72

Reynolds, J.C. "Definitional Interpreters for Higher-Order Programming Languages." *Proceedings of the ACM Annual Conference.* New York: ACM, Aug. 1972.

Abstract: *Higher-order programming languages (i.e., languages in which procedures or labels can occur as values) are usually defined by interpreters which are themselves written in a programming language based on the lambda calculus (i.e., an applicative language such as pure LISP). Examples include McCarthy's definition of LISP, Landin's SECD machine, the Vienna definition of PL/I, Reynolds' definitions of GEDANKEN, and recent unpublished work of L. Morris and C. Wadsworth. Such definitions can be classified according to whether the interpreter contains higher-order functions, and whether the order of the application (i.e., call-by-value versus call-by-name) in the defined language depends on the order of application in the defining language. As an example, we consider the definition of a simple applicative programming language by means of an interpreter written in a similar language. Definitions in each of the above classifications are derived from one another by informal but constructive methods. The treatment of imperative features such as jumps and assignment is also discussed.*

Ritchie74

Ritchie, D.M. and K.L. Thompson. "The UNIX Time-Sharing System." *Comm. ACM 17*, 7 (July 1974), 365-375.

Abstract: *UNIX is a general-purpose, multi-user, interactive operating system for the Digital Equipment Corporation PDP-11/40 and 11/45 computers. It offers a number of features seldom found even in larger operating systems, including: (1) a hierarchical file system incorporating demountable volumes; (2) compatible file, device, and interprocess I/O; (3) the ability to initiate asynchronous processes; (4) system command language selectable on a per-user basis; (5) over 100 subsystems including a dozen languages. This paper discusses the nature and implementation of the file system and of the user command interface.*

Robinson79

Robinson, L. *The HDM Handbook, Volume I: The Foundations of HDM.* SRI Project 4828, Computer Science Laboratory, SRI International, Menlo Park, CA, June 1979.

(From the Introduction) HDM provides an integrated collection of languages and tools that aid in the software development process. HDM addresses many of the aspects of the general software problem — namely that software is often late, too costly, unreliable, and noncompliant with its requirements....

In developing HDM, we have selected some particularly useful concepts and integrated them into a unified approach that encourages software developers to think about software development in terms of these concepts. This approach defines a system as consisting of a set of components arranged in a particular structure. The components are specified using languages developed for that purpose. Some properties of the specifications can be evaluated by on-line tools; others can be measured by subjective evaluation. The languages and tools of HDM have been designed to enforce its concepts and to realize its mechanisms.

This volume describes the basic concepts of HDM. The stages provide a suggested ordering of system development. Guidelines for the use of HDM are also presented.

Rolph

Rolph, S. and T. Alfano. *Statemate by Example.* Burlington, MA: i-Logix, date unknown.

This book shows by use of a simplified, but real example how STATEMATE might be used to design a reactive system. (The book has *no* sign of any publication date!)

Rombach87

Rombach, H.D. *Software Specification: A Framework.* Curriculum Module SEI-CM-11, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa., Oct. 1987.

(From Capsule Description) This module provides a framework for specifying software processes and products. The specification of a software product type describes how the correspoding products should look. The specification of a sofwtare process type describes how the corresponding processes should be performed.

Ruggiero79

Ruggiero, W., G. Estrin, R. Fenchel, R. Razouk, D. Schwabe, and M. Vernon. "Analysis of Data Flow Models Using the SARA Graph Model of Behavior." *National Computer Conference.* Montvale, NJ: AFIPS, June 1979.

Rushby91a

Rushby, J., F. von Henke, and S. Owre. *An Introduction to Formal Specification and Verification Using EHDM.* CSL Technical Report, SRI-CSL-91-02, Computer Science Laboratory, SRI International, Menlo Park, CA, Aug. 1991.

Abstract: *This report is a tutorial on formal specification and verification using EHDM. The EHDM specification language is very expressive, based on a strongly typed higher-order logic, enriched with elements of the Hoare (relational) calculus. The type system provides subtypes, dependent types, and*

certain forms of type-polymorphism. Modules are used to structure large specifications and support hierarchical development. The language has a complete formal semantic characterization and is supported by a fully mechanized specification and verification environment that has been used to develop large specifications and perform very hard formal verifications.

The tutorial uses simple examples to describe the EHDM language, methodology, and tools. The first examples illustrate the basic ideas of specification and theorem proving in EHDM. We then introduce the ideas of testing specifications, of horizontal and vertical hierarchy, and of consistency and conservative extension. Later chapters cover more advanced topics including subtypes, higher-order logic, proofs by induction, and program verification using Hoare logic. The tutorial is illustrated throughout with self-contained examples of EHDM specifications and proofs, all of which have been mechanically checked.

Rushby91b

Rushby, J. and F. von Henke. *Formal Verification of the Interactive Convergence Clock Synchronization Algorithm.* CSL Technical Report, SRI-CSL-89-3R, Computer Science Laboratory, SRI International, Menlo Park, CA, Aug. 1991.

Abstract: *We describe a formal specification and a mechanically checked verification of the Interactive Convergence Clock Synchronization Algorithm of Lamport and Melliar-Smith .... In the course of this work, we discovered several technical flaws in the analysis given by Lamport and Melliar-Smith, even though their presentation is unusually precise and detailed. As far as we know, these flaws (affecting the main theorem and four of its five lemmas) were not detected by the "social process" of informal peer scrutiny to which the paper has been subjected since its publication. We discuss the flaws in the published proof and give a revised presentation of the analysis that not only corrects the flaws in the original, but is also more precised and, we believe, easier to follow. This informal presentation was derived directly from our formal specification and verification. Some of our corrections to the flaws in the original require slight modifications to the assumptions underlying the algorithm and to the constraints on its parameters, and thus change the external specification of the algorithm.*

The formal analysis of the Interactive Convergence Clock Synchronization Algorithm was performed using the EHDM formal specification and verification environment. This application of EHDM provides a demonstration of some of the capabilities of the system.

Rushby91c

Rushby, J. *Formal Specification and Verification for Critical Systems: Tools, Achievements, and Prospects.* Computer Science Laboratory, SRI International, Menlo Park, CA, Aug. 1991.

Abstract: *Formal specification and verification use mathematical techniques to help document, specify, design, analyze, or certify computer software and hardware. Mathematically-based notation can provide specifications that are precises [sic] and unambiguous and that can be checked mechanically for certain types of error. Formal verification uses theorem proving techniques to establish consistency between one level of formal specification and another.*

This paper describes some of the issues in the design and use of formal specification languages and verification systems, outlines some examples of the application of formal methods to critical systems, and identifies the benefits that may be obtained from this technology.

Scheid83a

Scheid, J. *Implementation Specification.* TM-7315/000/00, System Development Corporation, Santa Monica, CA, 1983.

(From the Introduction) The Implementation Specification provides the capability for the user to express the connection between an abstract Ina Jo specification and the implementation of it in a Higher-Order Language (HOL) code.

Scheid83b

Scheid, J. *The Design of the Ina Jo Verification Condition Generator (VCG) for Modula.* TM-7393/000/00, System Development Corporation, Santa Monica, CA, 1983.

(From the Introduction) This document contains the functional design of the Ina Jo verification condition generator (VCG) for the Modula I programming language. Although the VCG can be used only for programs written in York Modula for PDP-11/Unix [sic] systems, much of the design is also applicable to other compilers / languages / operating systems / computers. One reason for this partial design independence is the use of a modified version of Ina Jo (Inamod) on both sides of the Ina Jo/VCG interface, i.e. in the implementation specifications and imbedded [sic] in the Modula code [sic].

Scheid86a

Scheid, J., S. Anderson, R. Martin, and S. Holtsberg. *The Ina Jo Specification Language Reference Manual.* TM-(L)-6021/001/02, SDC, A Burroughs Company, Santa Monica, CA, Jan. 1986.

Scheid86b
Scheid, J. and S. Holtsberg. *Enhancements to Formal Development Methodology (FDM): Ina Jo Definition.* TM-7527/016/00, SDC, A Burroughs Company, Santa Monica, CA, March 1986.

Scheid89
Scheid, J. and S. Holtsberg. *The Ina Jo Specification Language Reference Manual.* TM-(L)-6021/001/05, Unisys Corporation, Culver City, CA, May 1989.

(From the Introduction) Ina Jo is the specification language of the Formal Development Methodology (FDM). This reference manual describes Ina Jo as it is implemented in Release 12.4 of the FDM tools.

Schmidt86
Schmidt, D.A. *Denotational Semantics, A Methodology for Language Development.* Boston: Allyn and Bacon, 1986.

This book presents the topic of denotational semantics from an engineering standpoint, focusing on programming language description and implementation. Chapter 12 covers denotational semantics of nondeterminism and concurrency.

Schwabe85
Schwabe, D. and A.R. Cavalli. "Temporal Logic Specification of a Virtual Ring Lan Access Protocol." In *Protocol Specifications, Testing, and Verification, IV,* Y. Yemini, R. Strom, and S. Yemini, eds. Amsterdam: North-Holland, 1985.

Abstract: *This paper presents the use of temporal logic for the specification of the access protocol of a local area network, REDPUC, developed at the Department of Informatics of the Catholic University in Rio de Janeiro. The particular temporal logic system used allows the application of the automated proof techniques developed ... [by Cavalli* et al*]. The protocol described here exhibits a higher degree of complexity than other protocols previously described in the literature, especially if one considers only efforts using temporal logic.*

Schwartz81
Schwartz, R.L. and P.M. Melliar-Smith. "Temporal Logic Specification of Distributed Systems." *Second International Conference on Distributed Computing Systems.* Washington, DC: IEEE Computer Society, April 1981, 446-454.

Abstract: *This paper describes the use of temporal logic to specify protocols for distributed network communications. The Alternating Bit protocol, chosen for illustration, provides a simple yet non-trivial example of the method. Temporal logic lends a uniform framework in which to specify and formally verify both safety and progress (liveness) properties of the protocol.*

Shostak82
Shostak, R.E., R.L. Schwartz, and P.M. Melliar-Smith. "STP: A Mechanized Logic for Specification and Verification." In *Proceedings of the Sixth Conference on Automated Deduction.* Lecture Notes in Computer Science, no. 138. Berlin: Springer-Verlag, 1982.

(From the Introduction) This report describes a logic and proof theory that has been mechanized and successfully applied to prove nontrivial properties of a fully distributed fault-tolerant system. We believe the system is closer to achieving the critical balance in a man-machine interaction necessary for successful application by users other than the system developers.

STP is an implemented system supporting specification and verification of theories expressed in an extension of multisorted first-order logic. The logic includes type parameterization and type hierarchies. STP support includes syntactic checking and proof components as part of an interactive environment with a certain *core theory* that comprises a set of primitive types and function symbols, and extends this theory by introducing new types and symbols, together with axioms that capture the intended complete decision procedure for a certain syntactically characterizable subtheory. By providing aid to this component in the form of the selection of appropriate instances of axioms and lemmas, the user raises the level of competence of the prover to encompass the extended theory in its entirety. As a result of a successful proof attempt using STP, one obtains the sequence of intermediate lemmas, together with the axioms, auxiliary lemmas, and their necessary instantiations, which lead to the theorem. The system automatically keeps track of which formulas have been proved and which have not, so that the user is not forced to prove lemmas in advance of their application. The system also monitors the incremental introduction and modification of specifications to maintain soundness.

Silverberg79
Silverberg, B., L. Robinson, and K.N. Levitt. *The HDM Handbook, Volume II: The Languages and Tools of HDM.* SRI Project 4828, Computer Science Laboratory, SRI International, Menlo Park, CA, June 1979.

(From the Introduction) In this volume, we present the languages and tools of the SRI Hierarchical Development Methodology (HDM). The languages provide a way of recording and communicating de-

cisions made throughout stages of system design, specification, implementation, and verification. The tools assist the system developer during this development process. The current set of tools is used primarily to determine whether certain well-formedness and consistency criteria are satisfied.

The languages of HDM are intended to capture the concepts and computational model described in Volume I. SPECIAL (SPECIfication and Assertion Language) is used to specify modules and mapping functions. HSL (Hierarchy Specification Language) is used to describe the structuring of modules into machines, and machines into systems. ILPL (Intermediate Level Programming Language) is used to record module implementation decisions. In addition, the final implementation code is written in some executable programming language such as Pascal, Euclid, Ada, etc. Such implementation languages could also be considered "languages of HDM", though we will take a narrower view and restrict our attention to SPECIAL, HSL, and HLPL.

Smith85

Smith, M.K. and R.M. Cohen. "Gypsy Verification Environment: Status." *ACM Software Eng. Notes 10*, 4 (April 1985).

(From the Introduction) The Gypsy methodology is an integrated system of methods, languages, and tools for designing and building formally verified software systems. The methods provide for the specification and coding of programs that can be rigorously verified by logical deduction always to run according to specification. These specification, programming, and verification methods dictated the design of the program description language Gypsy.... Gypsy consists of two intersection components: a formal specification language and a verifiable, high level programming language. These component languages can be used separately or collectively. The methodology makes use of the *Gypsy Verification Environment* (GVE) to provide automated support. The GVE is a large interactive system that maintains a Gypsy program description library and provides a highly integrated set of tools for implementing the specification, programming, and verification methods.

Smith86

Smith, G. and D.V. Schorre. *The Interactive Theorem Manual (ITP) User's Manual.* TM-(L)-6889/000/006, SDC, A Burroughs Company, Santa Monica, CA, Dec. 1986.

(From the Introduction) SDC's Formal Development Methodology (FDM) is an integrated methodology for the design, specification, implementation and verification of software. The *Ina Jo* specification language forms the basis for this methodology, and formal verification of the specifications written in the *Ina Jo* language is accomplished by using FDM's Interactive Theorem Prover (ITP).

Smyth78

Smyth, M.B. "Powerdomains." *J. Comp. and Syst. Sci. 17* (1978), 23-36.

(From the Introduction) If the meaning of a deterministic program may be considered to be a function from D to D, where D is some domain of "states", then it would seem that the meaning of a nondeterministic program is a function from D to 2^D, or perhaps from 2^D to 2^D. To apply the methods of fix-point semantics, then, we should find some way to construe the power set of a domain as itself a domain, with a suitable ordering.

Stein80

Stein, J. and D.V. Schorre. *The Interactive Theorem Manual (ITP) User Manual.* TM-(L)-6889/000/001, System Development Corporation, Santa Monica, CA, Dec. 1980.

(From the Introduction) The interactive theorem prover (ITP) uses the rule of mathematical logic to generate, with the active and occasionally imaginative help of a human operator, proofs of complex theorems derived from specifications written in the *INA JO language. These proofs would be* extremely laborious if done without mechanized tools as the theorems are usually quite long and many proof steps are required. The ITP uses the principle of reductio as absurdum (or indirect derivation) to prove the theorems. To show that a particular set of conditions are true, the assumption is made that they are *not* true. From this assumption, contradictions (statements that negate each other) are derived; therefore, the original negated statements are contradictions and the original set of conditions are true.

Sunshine82

Sunshine, C.A., D.D. Thompson, R.W. Erickson, S.L. Gerhart, and D. Schwabe. "Specification and Verification of Communication Protocols in AFFIRM Using State Transition Models." *IEEE Trans. Software Eng. SE-8*, 5 (1982), 460-489.

Abstract: *It is becoming increasingly important that communication protocols be formally specified and verified. This paper describes a particular approach—the state transition model—using a collection of mechanically supported specification and verification tools incorporated in a running system called AFFIRM. Although developed for the specification of abstract data types and the verification of their properties, the formalism embodied in AFFIRM can also express the concepts underlying*

state transition machines. Such models easily express most of the events occurring in protocol systems, including those of the users, their agent processes, and the communication channels. The paper reviews the basic concepts of state transition models and the AFFIRM formalism and methodology and describes their union. A detailed example, the alternating bit protocol, illustrates various properties of interest for specification and verification. Other examples explored using this formalism are briefly described and the accumulated experience is discussed.

The paper is an excellent introduction to AFFIRM, and a demonstration of its practical utility in a realistic problem domain.

Tanenbaum87

Tanenbaum, A.S. *Operating Systems: Design and Implementation*. Englewood Cliffs, NJ: Prentice-Hall, 1987.

This book describes operating systems in general via the construction of MINIX, a UNIX look-alike that runs on IBM-PC compatibles. The book contains a complete MINIX manual and a complete listing of its C code.

Thompson81

Thompson, D.H. and R.W. Erickson. *AFFIRM Reference Manual*. USC Information Sciences Institute, Marina del Rey, CA, Feb. 1981.

Abstract: *Affirm is an experimental interactive system for the development of specifications and the verification of abstract data types and algorithms. This document discusses the major concepts behind Affirm, and explains the purpose and use of each of the abstract machines comprising the structure of the system as seen by the user.*

Vernon80

Vernon, M., W. Overman, and R. Razouk. *GMB PL1 Preprocessor Reference Manual*. Computer Science Department, University of California, Los Angeles, CA, Jan. 1980.

Wegner68

Wegner, P. *Programming Languages, Information Structures, and Machine Organization*. New York: McGraw-Hill, 1968.

This book introduces the concept of Information Structure Model and uses it to describe programming languages and computers.

Wegner70

Wegner, P. "Three Computer Cultures: Computer Technology, Computer Mathematics, and Computer Science." In *Advances in Computers*. New York: Academic Press, 1970, 7-78.

Abstract: *Computers have proved so useful as scientific and technical tools that computer science is widely regarded as a technological discipline whose purpose is to create problem-solving tools for other disciplines. Within computer science there is a group of theoreticians who build mathematical models of computational processes. Yet computer science is neither a branch of technology nor a branch of mathematics. It involves a new way of thinking about computational schemes that is partly technological and partly mathematical, but contains a unique ingredient that differs qualitatively from those of traditional disciplines. This paper illustrates the special quality which distinguishes computer science from technology and mathematics by means of examples from the emerging theory of programming languages.*

Wegner72

Wegner, P. "The Vienna Definition Language." *ACM Computing Surveys 4*, 1 (March 1972), 5-63.

Abstract: *The Vienna Definition Language (VDL) is a programming language for defining programming languages. It allows us to describe precisely the execution of the set of all programs of a programming language. However, the Vienna Definition Language is important not only as one definition technique among many others but as an illustration of a new information-structure-oriented approach to the study of programming languages. This paper may be regarded as a case study in the information structure modeling of programming languages, as well as an introduction to a specific modeling technique....*

Wing89

Wing, J.M. and M. Nixon. "Extending Ina Jo with Temporal Logic." *IEEE Trans. Software Eng. SE-15*, 2 (Feb. 1989), 181-197.

Abstract: *Toward the overall goal of putting formal specifications to practical use in the design of large systems, we explore the combination of two specification methods: using temporal logic to specify concurrency properties and using an existing specification language, Ina Jo, to specify functional behavior of nondeterministic systems. In this paper, we give both informal and formal descriptions of both current Ina Jo and Ina Jo enhanced with temporal logic. We include details of a simple example to demonstrate the use of the proof system and details of an extended example to demonstrate the expressiveness of the enhanced language. We discuss*

at length our language design goals, decisions, and their implications. The appendix contains a list of axioms, rules of inference, derived rules, and theorem schemata for the enhanced formal system.

Wirth66
Wirth, N. and H. Weber. "EULER: A Generalization of ALGOL and its Formal Definition." *Comm. ACM* 9, 1 & 2 (January & February 1966), 13-23 & 89-99.

Abstract: *A method for defining programming languages is developed which introduces a rigorous relationship between structure and meaning. The structure of a language is defined by a phrase structure syntax, the meaning in terms of the effects which the execution of a sequence of interpretation rules exerts upon a fixed set of variables, called the Environment. There exists a one-to-one correspondence between syntactic rules and interpretation rules, and the sequence of executed interpretation rules is determined by the sequence of corresponding syntactic reductions which constitute a parse. The individual interpretation rules are explained in terms of an elementary and obvious algorithmic notation. A constructive method for evaluating a text is provided, and for certain decidable classes of languages their unambiguity is proved. As an example, a generalization of ALGOL is described in full detail to demonstrate that concepts like block-structure, procedures, parameters, etc. can be defined adequately and precisely by this method.*

Wolper82
Wolper, P. "Specification And Synthesis of Communicating Processes Using an Extended Temporal Logic." *Conference Record Ninth Annual ACM Symposium on Principles of Programming Languages.* New York: ACM, Jan. 1982, 20-33.

Abstract: *We apply an Extended Propositional Temporal Logic (EPTL) to the specification and synthesis of the synchronization part of communicating processes. To specify a process, we give an EPTL formula that describes its sequence of communications. The synthesis is done by constructing a model of the given specifications using a tableau-like satisfiability algorithm for the extended temporal logic. This model can then be interpreted as a program.*

Woodcock87
Woodcock, J.C.P. "Transaction Processing Primitives and CSP." *IBM Journal of Research and Development 31*, 5 (1987), 535-545.

Abstract: *Several primitives for transaction processing systems are developed using the notations of Communicating Sequential Processes. The approach taken is to capture each requirement separately, in the simplest possible context. The specification is then the conjunction of all these requirements. As each is developed as a predicate over traces of the observable events in the system, it is also implemented as a simple communicating processs; the implementation of the entire system is then merely the parallel composition of these processes. The laws of CSP are then used to transform the system to achieve the required degree of concurrency, to make it suitable for execution in a multiple-tasking system, for example. Finally, there is a discussion of how state-based systems may be developed using this approach together with some appropriate notation for specifying and refining data structures and operation upon them and of how the system may be implemented. This work is intended as a case study in the use of CSP.*

Woodward79
Woodward, J. "Applications for Multilevel Secure Operating Systems." *National Computer Conference.* Montvale, NJ: AFIPS, 1979, 319-328.

(From the Introduction) The need for secure computer systems has been identified in many areas of DoD operations, but in the past these systems have not been built in a secure manner because a secure operating system on which to run has not existed. Now that verifiably secure microcomputer operating systems are becoming a reality, applications for secure systems are becoming more clearly thought-out, designed and implemented. This paper surveys some proposed DoD and non-DoD secure computer applications.

Yu90
Yu, C.-F. and V.D. Gligor. "A Specification and Verification Method for Preventing Denial of Service." *IEEE Trans. Software Eng. SE-16*, 6 (1990), 581-592.

Abstract: *In this paper, we present a specification and verification method for preventing denial of service in the absence of failures and of integrity violations. We introduce the notion of "user agreements" and argue that lack of specifications for these arguments and for simultaneity conditions makes it impossible to demonstrate denial-of-service prevention, in spite of demonstrably fair service access. We illustrate the use of this method with an example and explain why current methods for specification and verification of safety and liveness properties of concurrent programs do not handle this problem. The proposed specification and verification method is meant to augment current methods for secure system design.*

Zave72
Zave, P. "An Operational Approach to Requirements Specification for Embedded Systems." *IEEE Trans. Software Eng. SE-8*, 3 (1972), 250-269.

Abstract: *The approach to requirements specification for embedded systems described in this paper is called 'operational' because a requirements specification is an executable model of the proposed system interacting with its environment. The approach is embodied by the language PAISLey, which is motivated and defined herein. Embedded systems are characterized by asynchronous parallelism, even at the requirements level; PAISLey specifications are constructed by interacting processes so that this can be represented directly. Embedded systems are also characterized by urgent performance requirements, and PAISLey offers a formal, but intuitive treatment of performance.*

Zave87a
Zave, P. *PAISLey User Documentation, Volume 1: REFERENCE MANUAL.* Computer Technology Research Laboratory, AT&T Bell Laboratories, 1987.

Zave87b
Zave, P. *PAISLey User Documentation, Volume 2: TUTORIAL.* Computer Technology Research Laboratory, AT&T Bell Laboratories, 1987.

(From Prologue) PAISLey is an *executable specification language*. It is fully formal and can be executed by an interpreter, just like any programming language. But it is also meant to be as implementation-independent as possible, so that it can describe the required properties and behavior of a digital system without constraining how those properties and behavior are implemented.

Zave87c
Zave, P. *PAISLey User Documentation, Volume 3: CASE STUDIES.* Computer Technology Research Laboratory, AT&T Bell Laboratories, 1987.

(From the Introduction) The purpose of this volume of documentation is to provide examples of PAISLey specifications. There are plenty of examples in the tutorial, but the specifications in this volume are different in two important ways: (1) Most of the examples in the tutorial are fragments of specifications selected to illustrate particular points about PAISLey. The specifications here are all described in their entirety, and they are presented so as to simulate the mental processes that created them. (2) I made up all the examples in the tutorial by myself, with no outside constraints. The case studies are exactly the opposite—each specification solves a problem that came from somewhere else, and I was forced to confront its difficulties rather than avoid or change them.

The case studies fit into two major categories, "Academic Problems" are all relatively simple, relatively unrealistic problems that have been posed for the benefit of researchers. Because of their simplicity, they are solved in great detail. "Real Systems" are just that—system-development projects on which PAISLey has been used. Due to their sizes the specifications of real systems are described in general rather than presented in detail.

Zave91
Zave, P. "An Insider's Evaluation of PAISLey." *IEEE Trans. Software Eng. 17*, 3 (March 1991), 212-225.

Abstract: *PAISLey is an executable specification language, accompanied by specification methods, analysis techniques, and software tools; it was the subject of a long-term research project. The paper also discusses research methods—both how the results were obtained, and how the project might have been improved.*

Tables and Figures

This section contains all the Tables and Figures cited in the text of the module.

Below, all distinct symbols are assumed to be unequal.
Also, τ is taken to represent a hidden event also in CSP.

CCS	CSP
$a.P$	$a \rightarrow P$
$(a.P)+(b.Q)$	$(a \rightarrow P \mid b \rightarrow Q)$
$(a.P)+(a.Q)$	$(a \rightarrow P) \prod (a \rightarrow Q)$
$(a.P)+(\bar{a}.Q)$	$\tau \rightarrow (P \mid Q) \equiv (P \mid Q)$
$(\tau.P)+(\tau.Q)$	$P \prod Q$
$(\tau\ P)+(a.Q)$	$P \prod (P \Box (a \rightarrow Q))$
NIL	*STOP*

Table 1

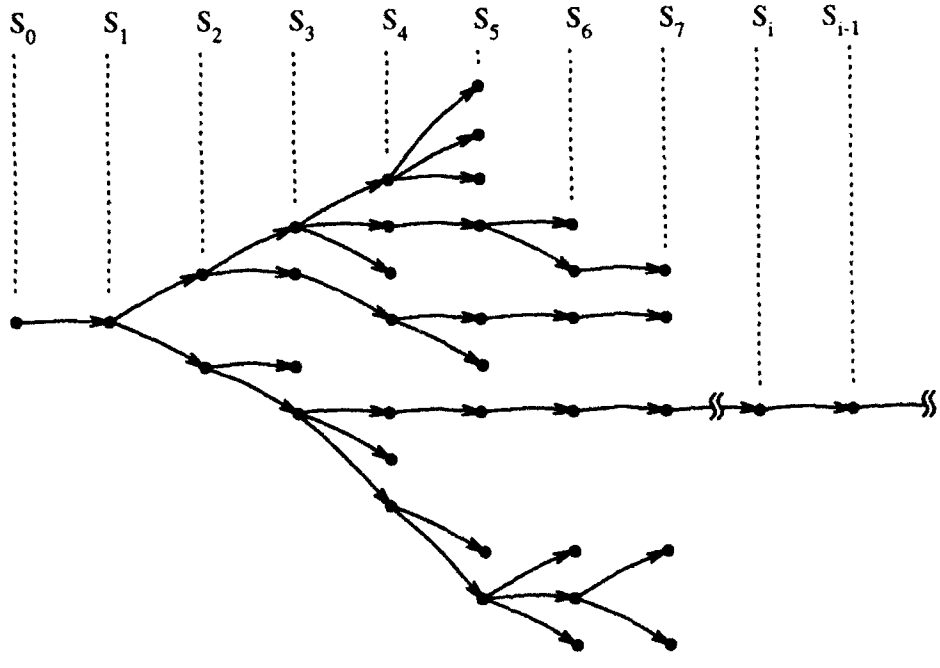


Figure 1

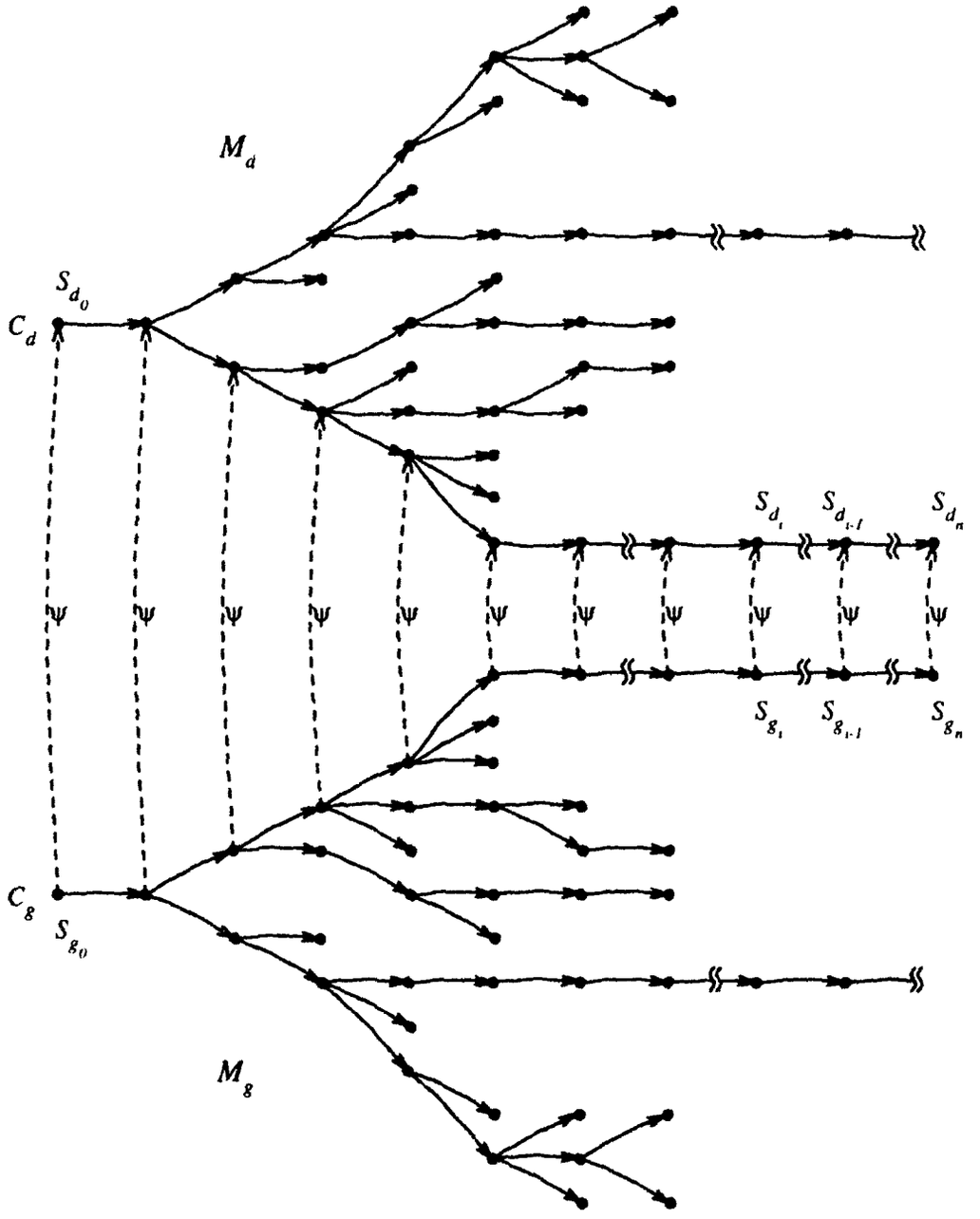


Figure 2

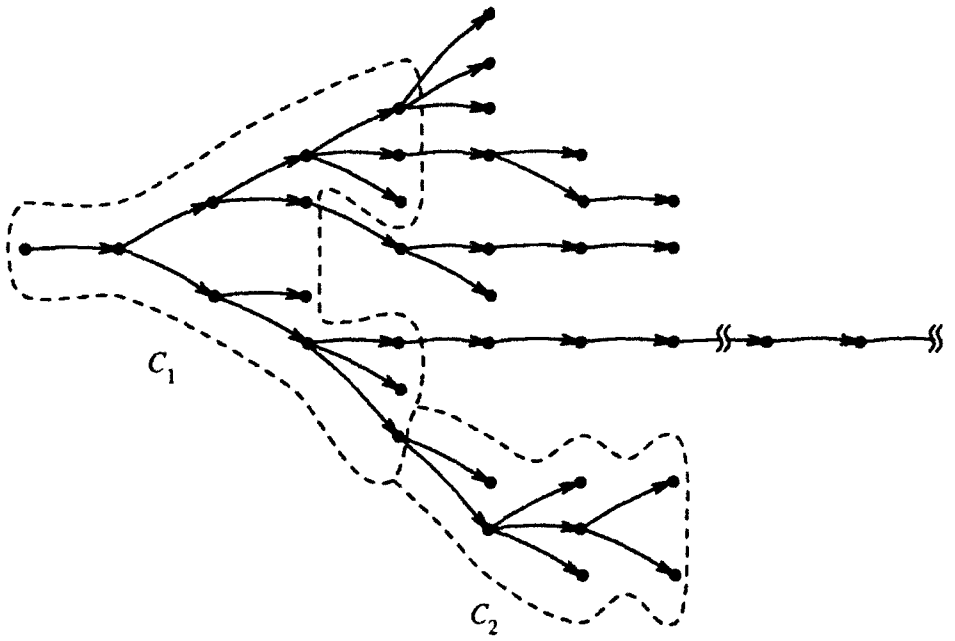


Figure 3

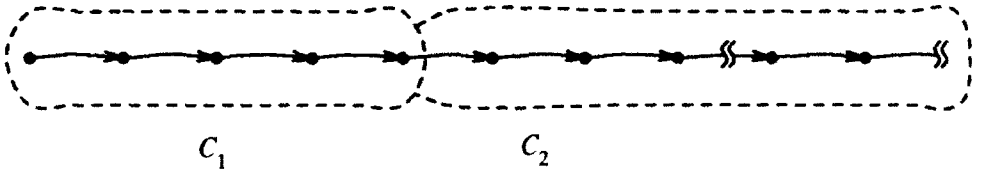


Figure 4

Approaches that can be used
to prove Properties about Programs

programs **properties**	functional	looping
safety	axiomatic temporal operational denotational	temporal operational
liveness	temporal denotational	temporal operational

Properties that can be proved about
Programs with available Approaches

programs **approaches**	functional	looping
axiomatic	safety	(prove only vacuous properties)
denotational	safety liveness	(treats as undefined)
operational	safety liveness	safety liveness
temporal	safety liveness	safety liveness

Programs for which Properties can
be proved with available Approaches

properties **approach**	safety	liveness
axiomatic	functional	(not possible)
denotational	functional	functional
operational	functional looping	functional looping
temporal	functional looping	functional looping

Figure 5

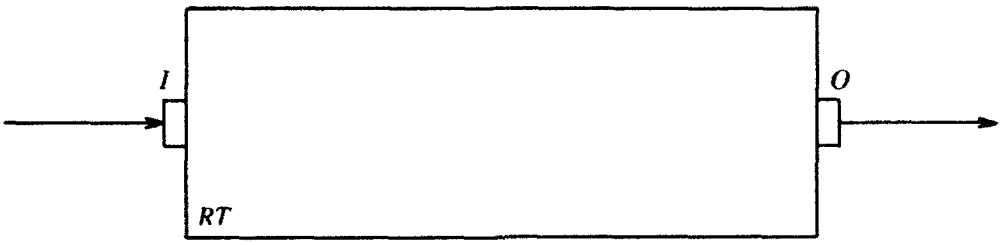


Figure 6

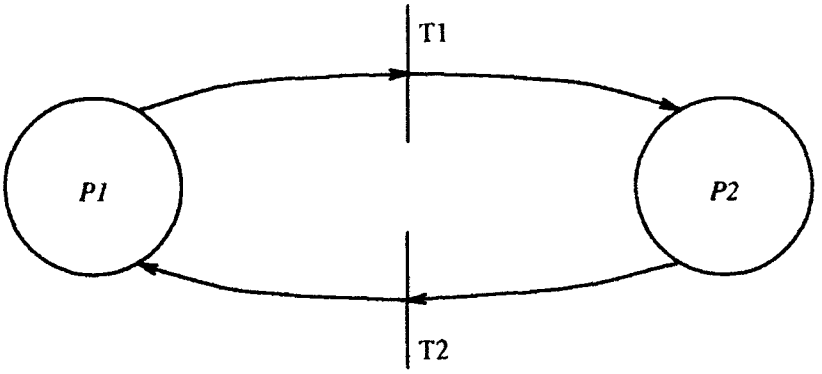


Figure 7

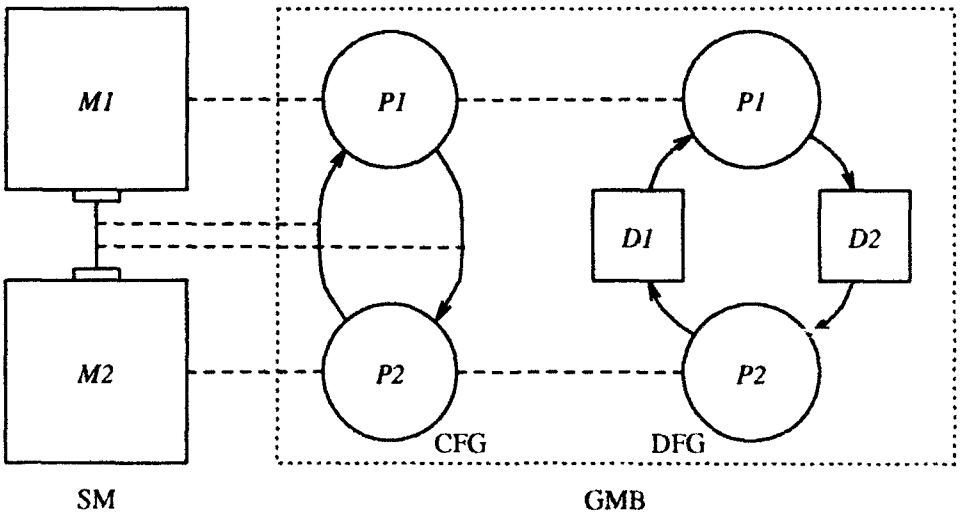


Figure 8

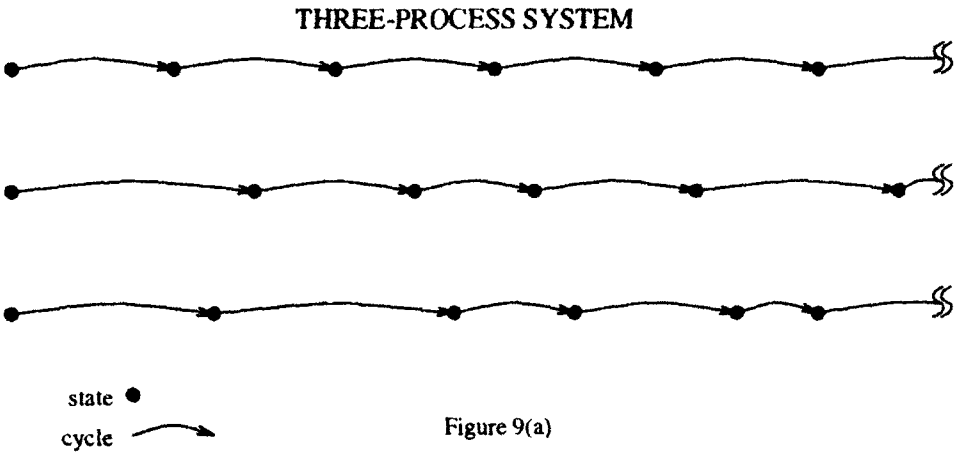


Figure 9(a)

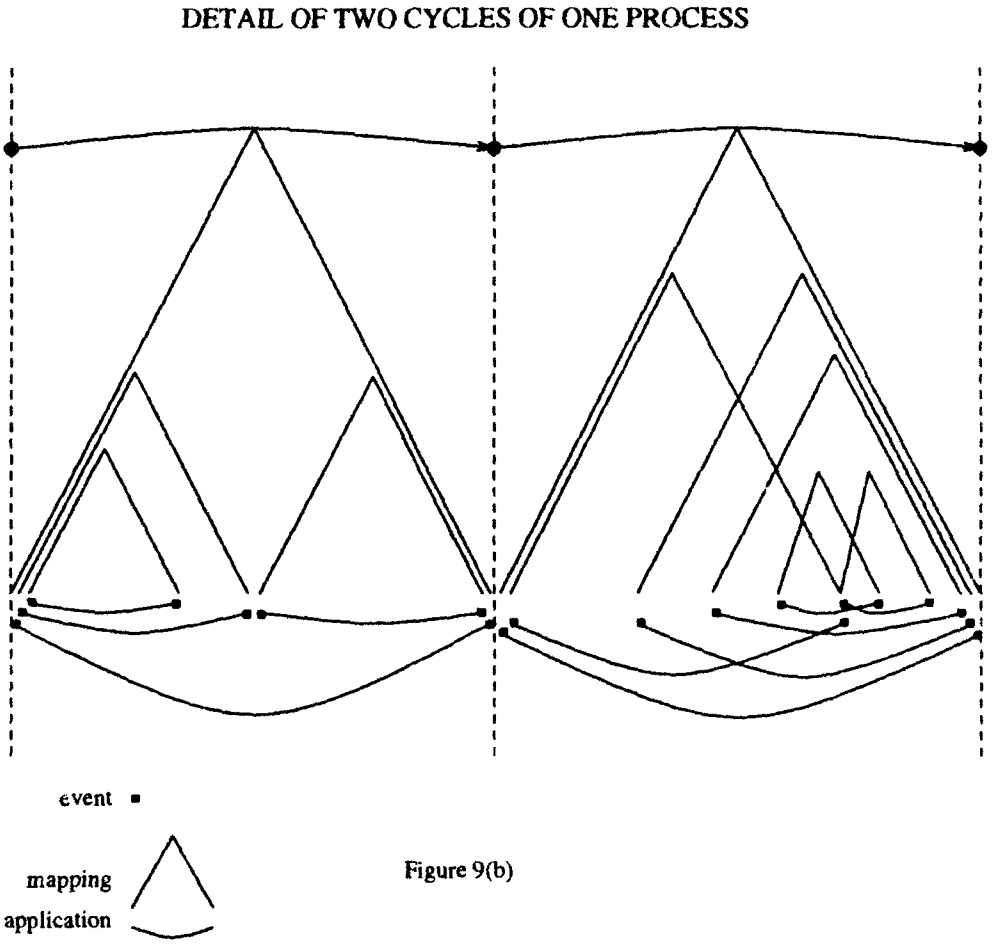


Figure 9(b)

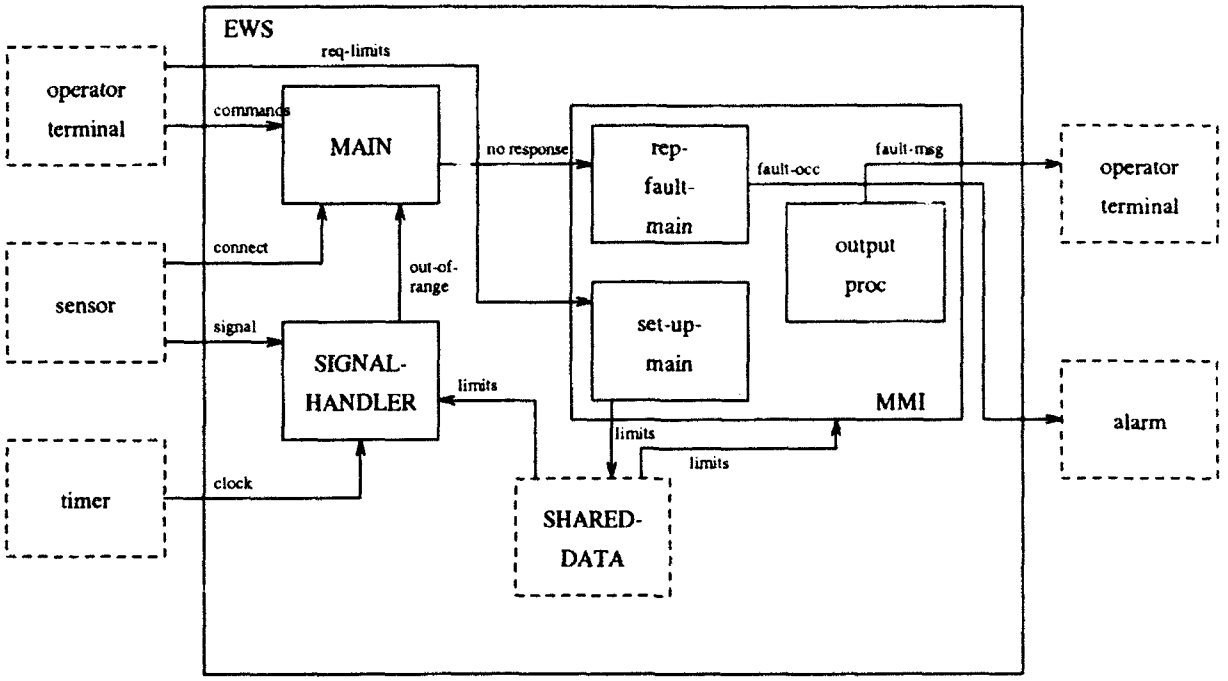

EWS
operator terminal
req-limits
commands
MAIN
no response
rep-fault-main
fault-occ
fault-msg
operator terminal
output proc
connect
out-of-range
sensor
signal
SIGNAL-HANDLER
set-up-main
limits
MMI
alarm
limits
limits
timer
clock
SHARED-DATA

Figure 10

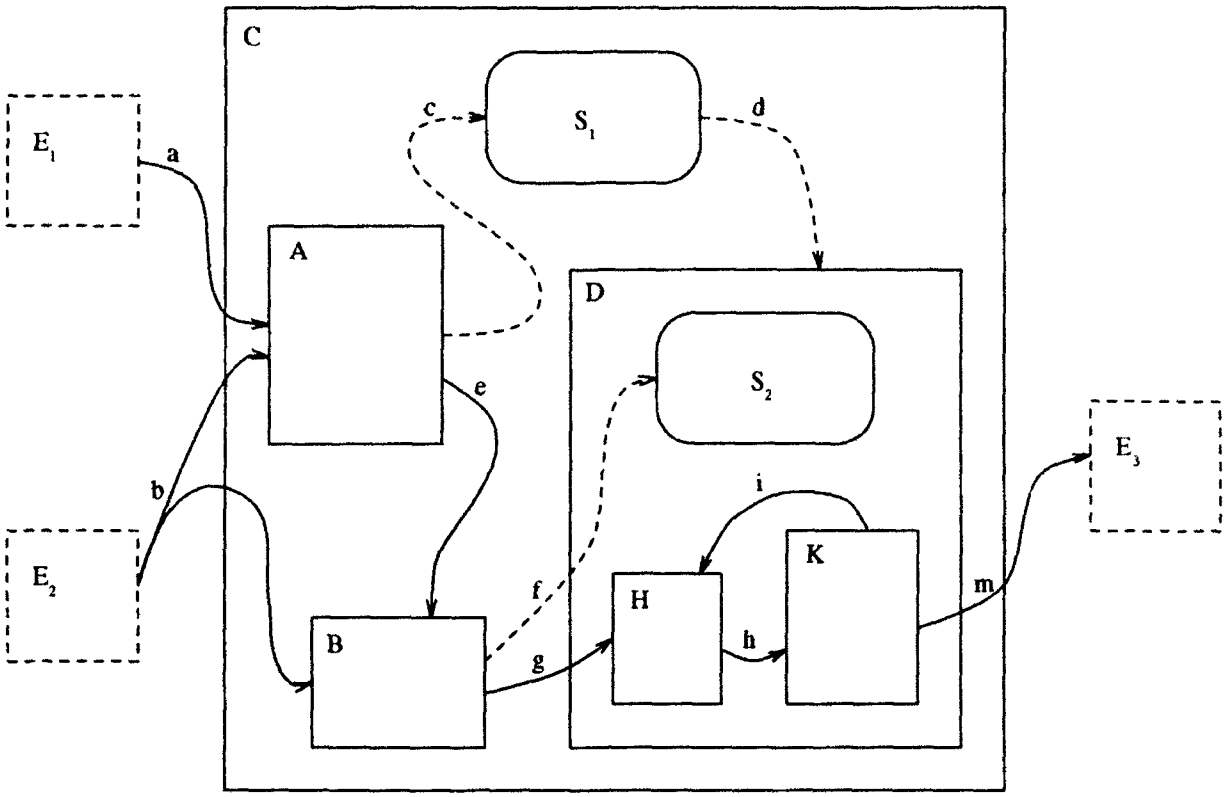
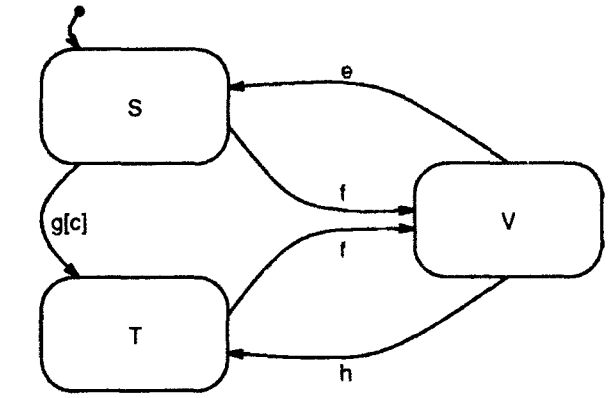
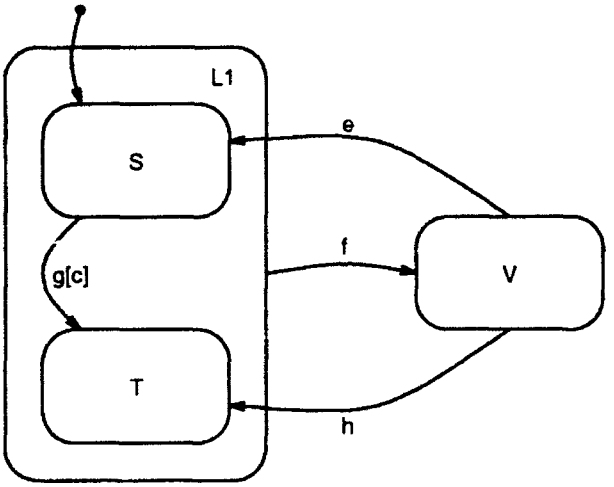


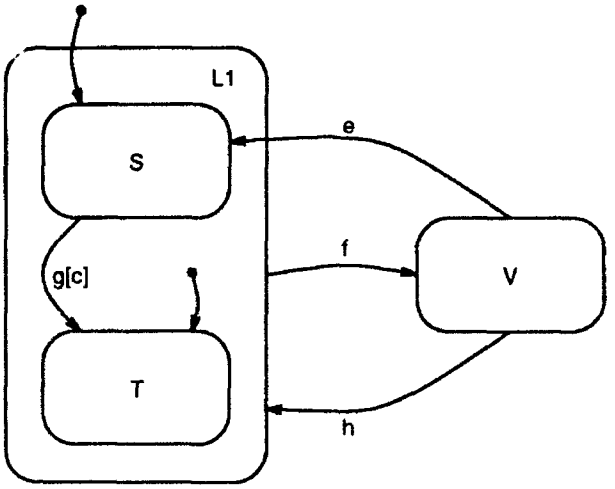
Figure 11



(a)

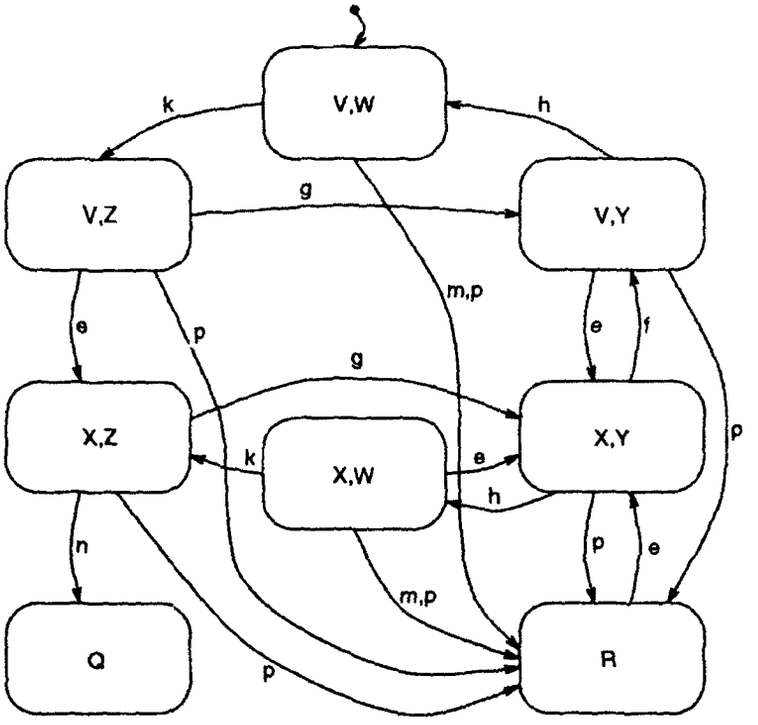


(b)

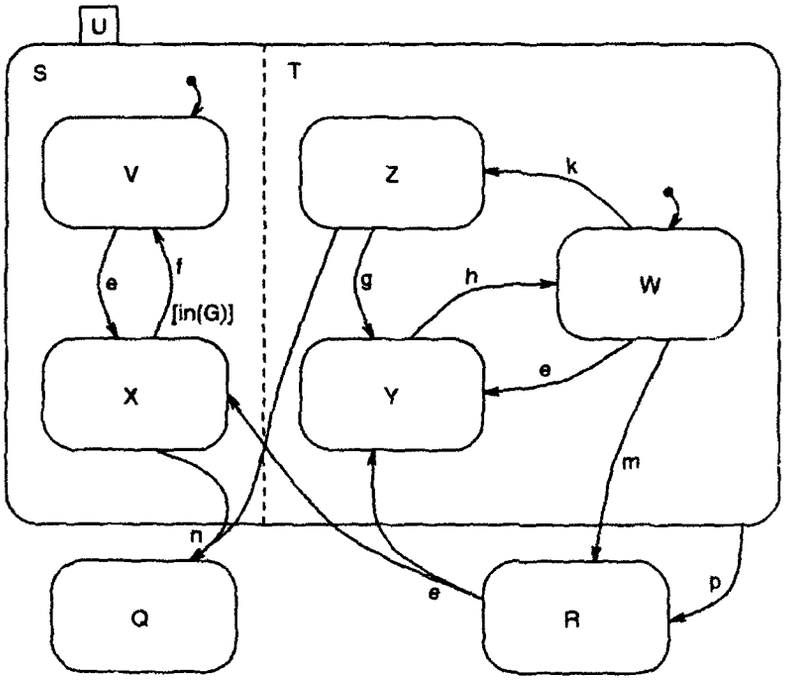


(c)

Figure 12


V,W
k
h
V,Z
g
V,Y
m,p
e
p
e
f
g
X,Z
k
X,W
e
X,Y
p
h
n
p
e
m,p
Q
p
R

(a)


U
S
T
V
Z
k
e
f
[in(G)]
g
h
W
X
Y
e
m
n
e
p
Q
R

(b)

Figure 13

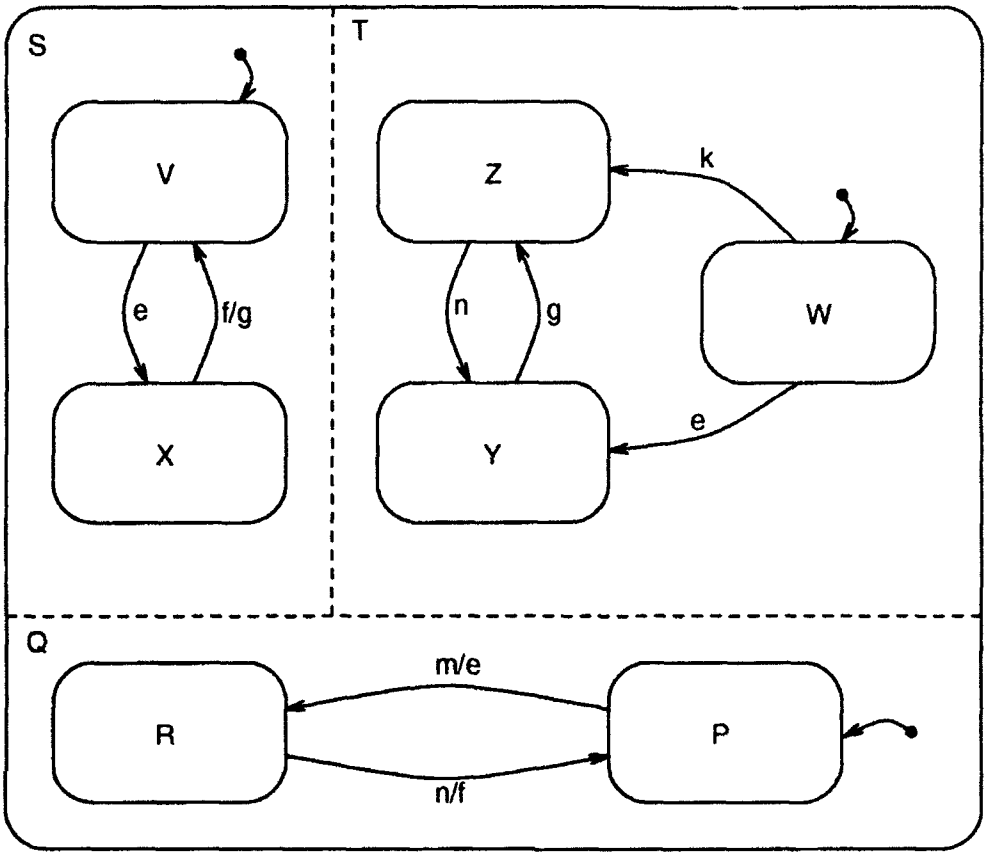


Figure 14

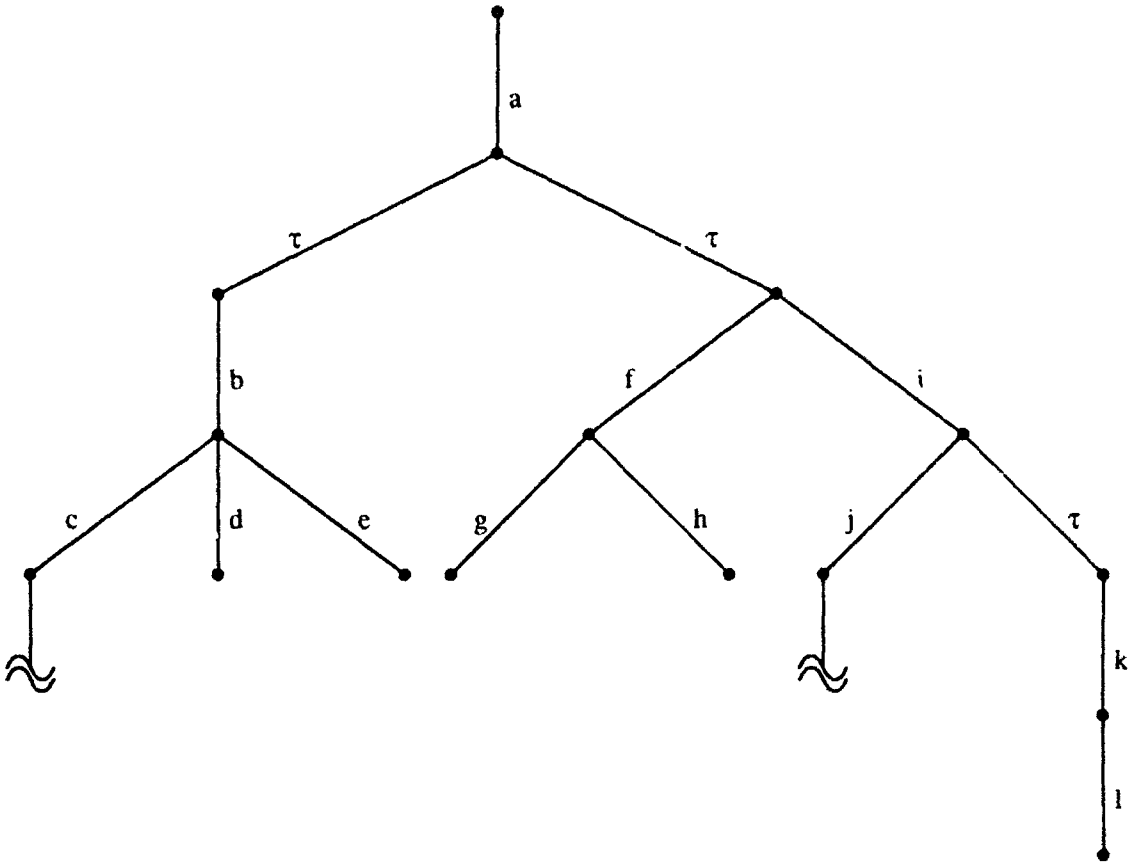


Figure 15

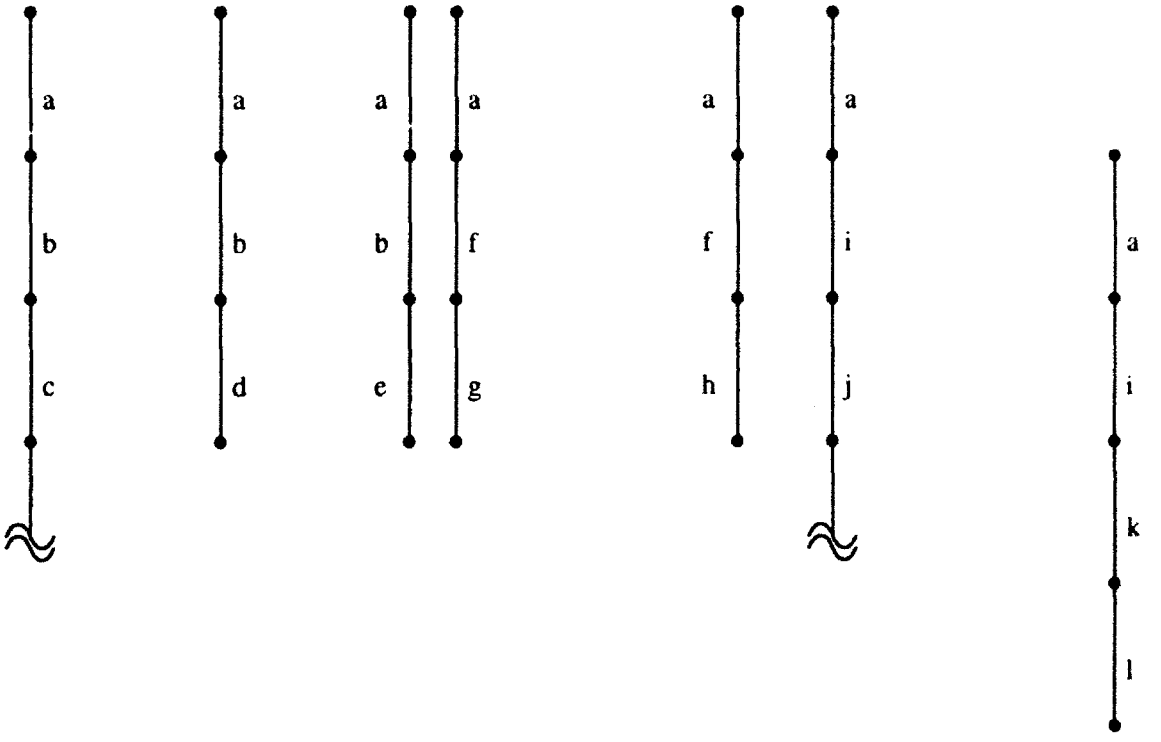


Figure 16

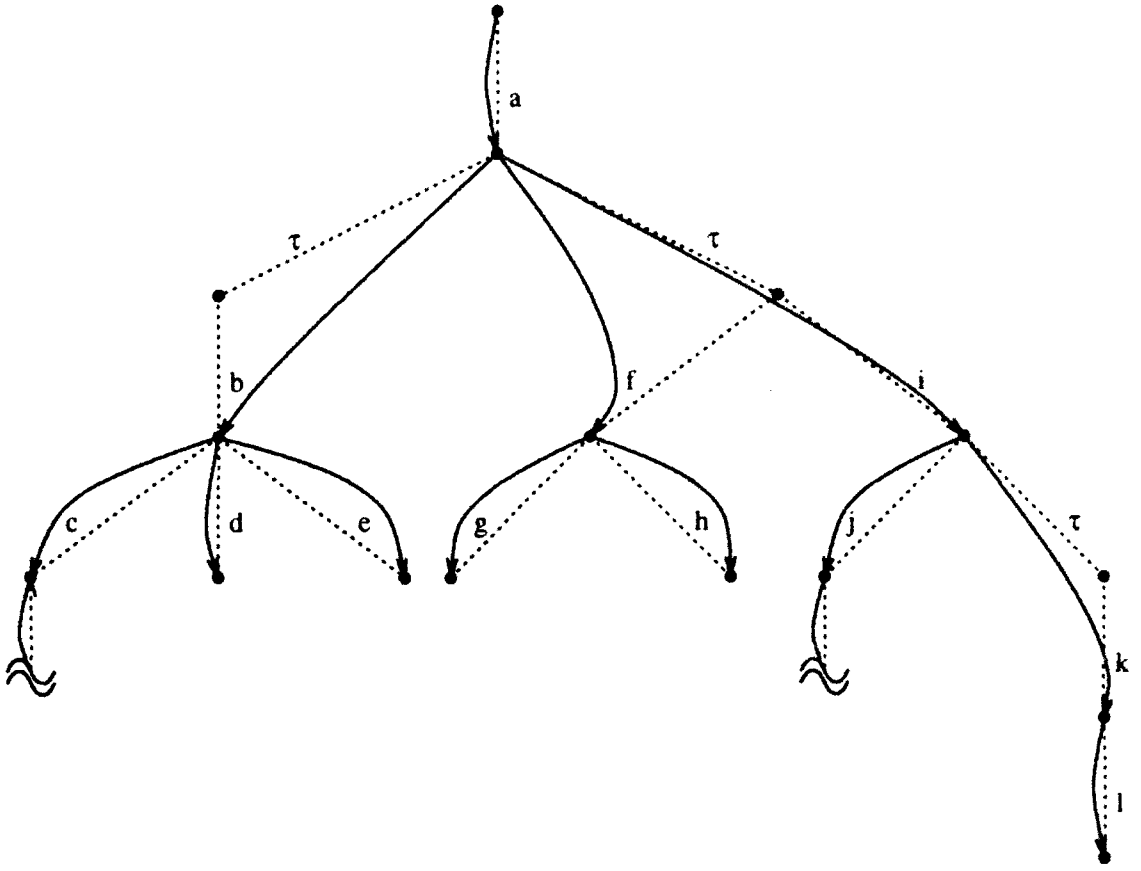


Figure 17

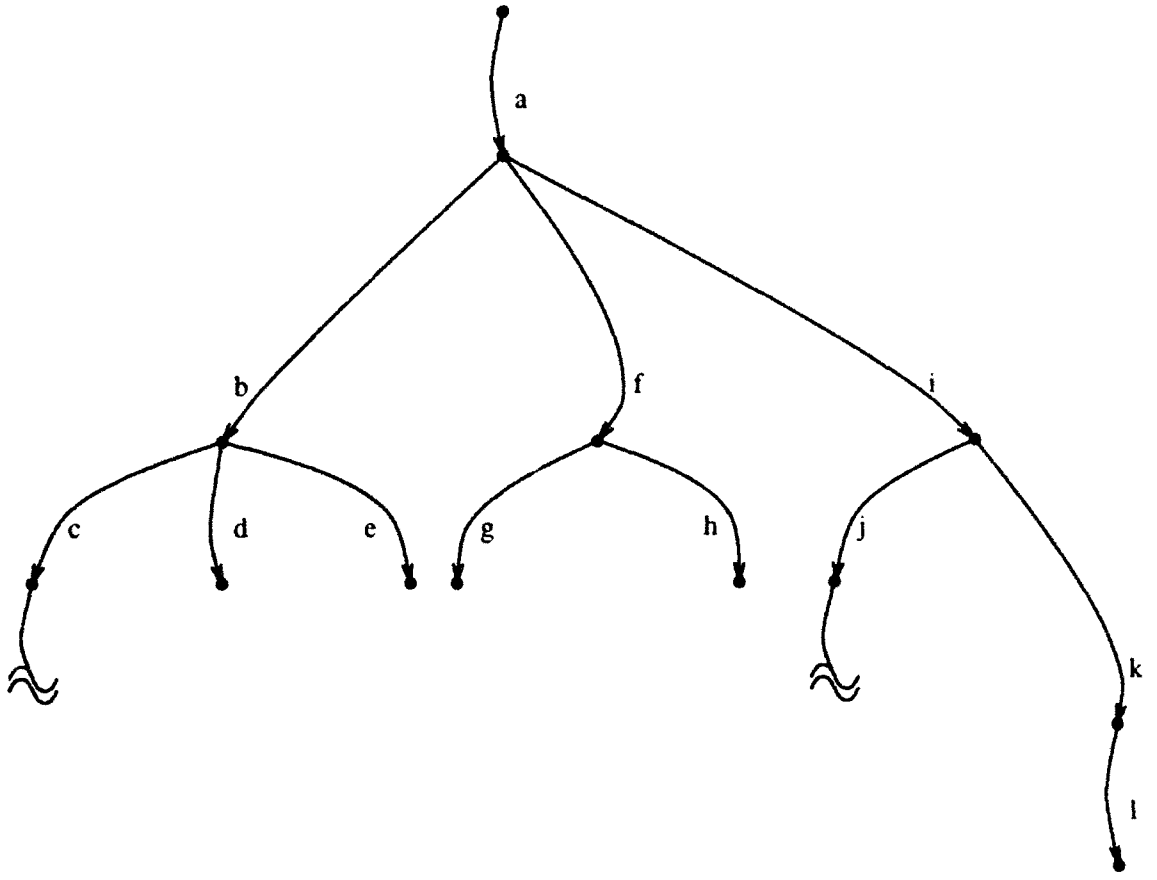


Figure 18

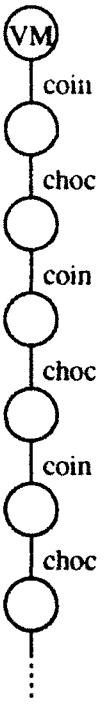


Figure 19

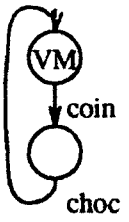


Figure 20

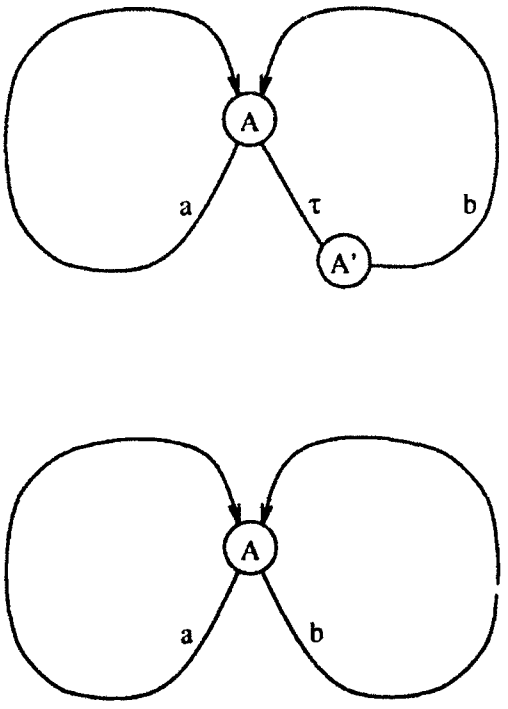


Figure 21

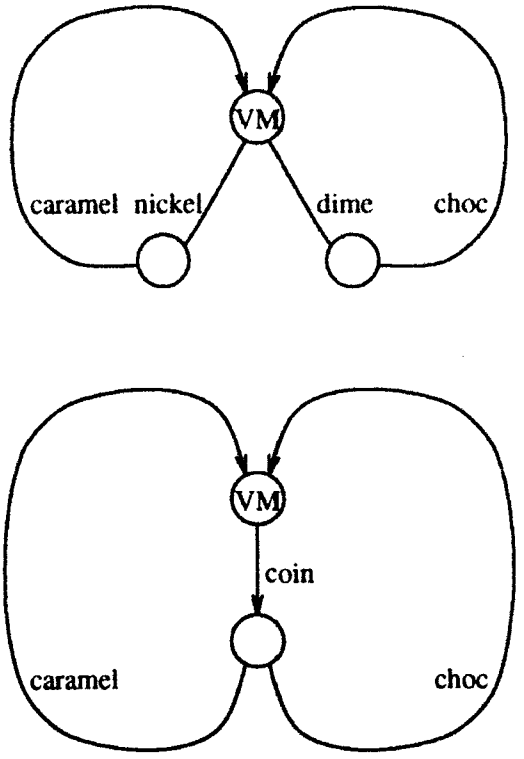


Figure 22

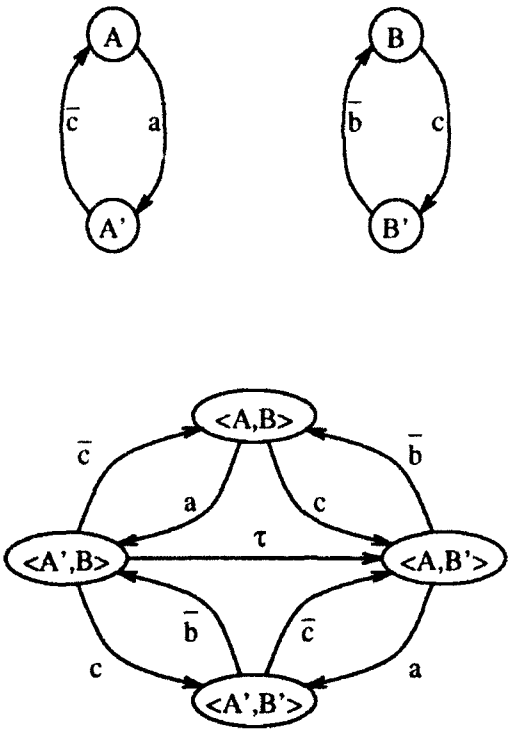


Figure 23

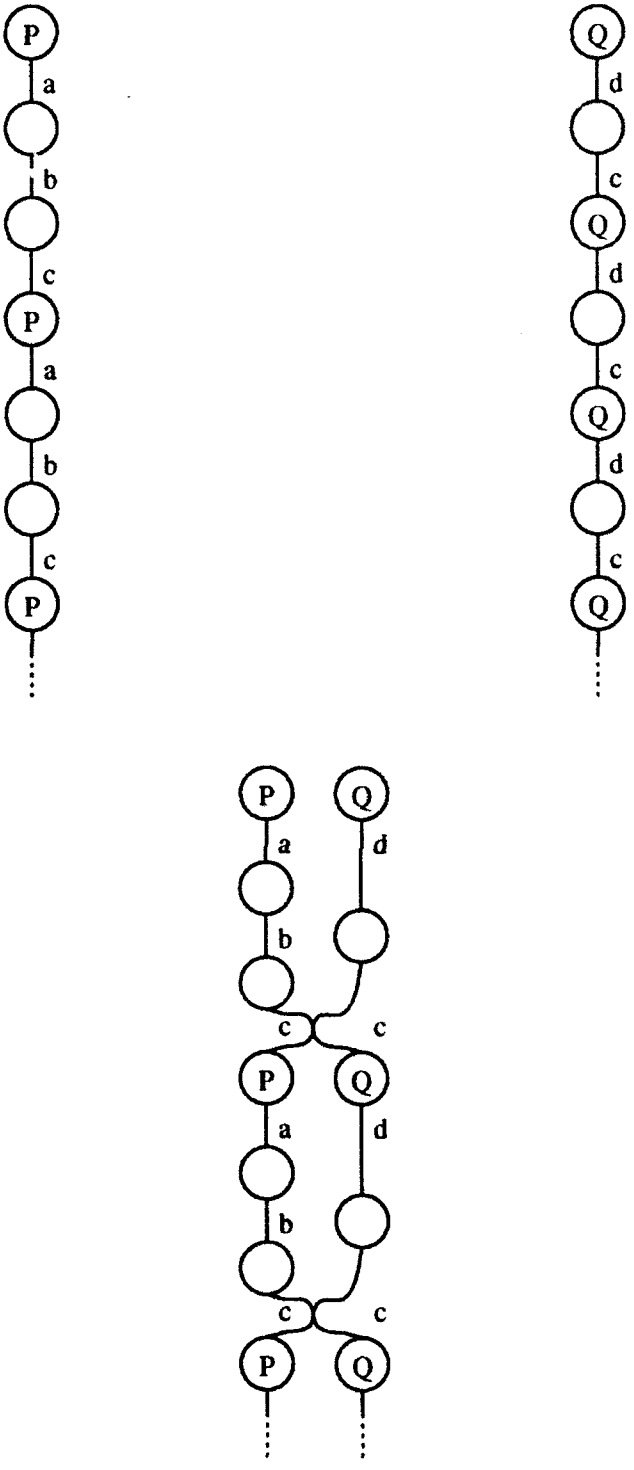


Figure 24

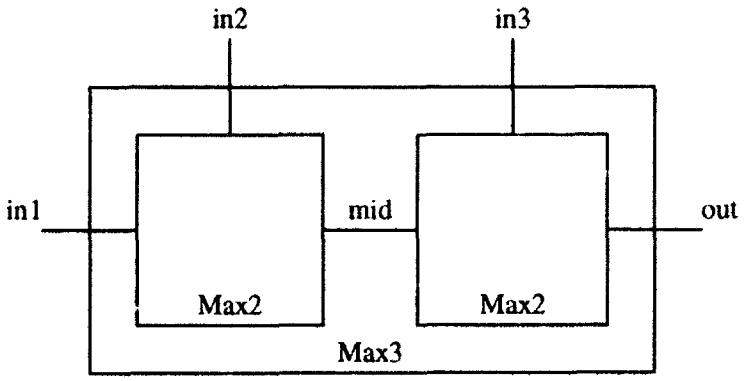


Figure 25

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS None
2a. SECURITY CLASSIFICATION AUTHORITY N/A	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for Public Release Distribution Unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) SEI-CM-27-1.0	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Software Engineering Institute 6b. OFFICE SYMBOL (if applicable) SEI	7a. NAME OF MONITORING ORGANIZATION SEI Joint Program Office
6c. ADDRESS (city, state, and zip code) Carnegie Mellon University Pittsburgh PA 15213	7b. ADDRESS (city, state, and zip code) ESC/AVS Hanscom Air Force Base, MA 01731
8a. NAME OFFUNDING/SPONSORING ORGANIZATION SEI Joint Program Office 8b. OFFICE SYMBOL (if applicable) ESC/AVS	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F1962890C0003
8c. ADDRESS (city, state, and zip code)) Carnegie Mellon University Pittsburgh PA 15213	10. SOURCE OF FUNDING NOS. PROGRAM ELEMENT NO 63756E; PROJECT NO. N/A; TASK NO N/A; WORK UNIT NO. N/A
11. TITLE (Include Security Classification) Formal Specification and Verification of Concurrent Programs SEi Curriculum Module	
12. PERSONAL AUTHOR(S) Daniel M. Berry	
13a. TYPE OF REPORT Final 13b. TIME COVERED FROM TO	14. DATE OF REPORT (year, month, day) February 1993 15. PAGE COUNT 101 pp.
16. SUPPLEMENTARY NOTATION	

17. COSATI CODES			18. SUBJECT TERMS (continue on reverse of necessary and identify by block number)	
FIELD	GROUP	SUB. GR.	concurrent programs	multiprocessing
			formal specification	multitasking
			formal verification	semantics of concurrency

19. ABSTRACT (continue on reverse if necessary and identify by block number)

This module introduces formal specification of concurrent software and verification of the consistency between concurrent programs and their specifications. First, what one might want to be able to prove about a concurrent program is discussed. Then, a number of formal descriptions of the concept are presented. These vary in their coverage of the phenomena, and some can be used as the bases of formal specification of programs. Next, techniques for carrying out the proof of consistency between the specification and the program are described. Finally, it is noted that some of these techniques have automated tools such as verifiers associated with them.

(please turn over)

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT UNCLASSIFIED/UNLIMITED ■ SAME AS RPT ☐ DTIC USERS ■	21. ABSTRACT SECURITY CLASSIFICATION Unclassified, Unlimited Distribution
22a. NAME OF RESPONSIBLE INDIVIDUAL Thomas R. Miller, Lt Col, USAF	22b. TELEPHONE NUMBER (include area code) (412) 268-7631 22c. OFFICE SYMBOL ESC/AVS (SEI)

ABSTRACT — continued from page one, block 19

The Software Engineering Institute (SEI) is a federally funded research and development center, operated by Carnegie Mellon University under contract with the United States Department of Defense.

The SEI Graduate Curriculum Project is developing a wide range of materials to support software engineering education. A *curriculum module* (CM) identifies and outlines the content of a specific topic area, and is intended to be used by an instructor in designing a course. A *support materials* package (SM) contains materials related to a module that may be helpful in teaching a course. An *educational materials* package (EM) contains other materials not necessarily related to a curriculum module. Other publications include software engineering curriculum recommendations and course designs.

SEI educational materials are being made available to educators throughout the academic, industrial, and government communities. The use of these materials in a course does not in any way constitute an endorsement of the course by the SEI, by Carnegie Mellon University, or by the United States government.

Comments on SEI educational materials and requests for additional information should be addressed to SEI Products, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania 15213. Electronic mail can be sent to education@sei.cmu.edu on the Internet.

Curriculum Modules (* Support Materials available)

CM-1 [superseded by CM-19]
CM-2 Introduction to Software Design
CM-3 The Software Technical Review Process*
CM-4 Software Configuration Management*
CM-5 Information Protection
CM-6 Software Safety
CM-7 Assurance of Software Quality
CM-8 Formal Specification of Software*
CM-9 Unit Analysis and Testing
CM-10 Models of Software Evolution: Life Cycle and Process
CM-11 Software Specifications: A Framework
CM-12 Software Metrics
CM-13 Introduction to Software Verification and Validation
CM-14 Intellectual Property Protection for Software
CM-15 [no longer available]
CM-16 Software Development Using VDM
CM-17 User Interface Development*
CM-18 [superseded by CM-23]
CM-19 Software Requirements
CM-20 Formal Verification of Programs
CM-21 Software Project Management
CM-22 Software Design Methods for Real-Time Systems
CM-23 Technical Writing for Software Engineers
CM-24 Concepts of Concurrent Programming
CM-25 Language and System Support for Concurrent Programming*
CM-26 Understanding Program Dependencies
CM-27 Formal Specification and Verification of Concurrent Programs

Educational Materials

EM-1 Software Maintenance Exercises for a Software Engineering Project Course
EM-2 APSE Interactive Monitor: An Artifact for Software Engineering Education
EM-3 Reading Computer Programs: Instructor's Guide and Exercises
EM-4 A Software Engineering Project Course with a Real Client
EM-5 Scenes of Software Inspections: Video Dramatizations for the Classroom
EM-6 Materials to Support Teaching a Project-Intensive Introduction to Software Engineering